PROGRAMMAZIONE DELLO Z80

EDIZIONE ITALIANA

Rodnay Zaks

SYBEX

GRUPPO EDITORIALE JACKSON

PROGRAMMAZIONE DELLO Z80

di
Rodnay Zaks

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

© Copyright per l'edizione originale Sybex Inc. 1979
© Copyright per l'edizione italiana Sybex Inc. 1981

Il Gruppo Editoriale Jackson ringrazia per il prezioso lavoro svolto nella stesura dell'edizione italiana le signore Francesca di Fiore, Rosi Bozzolo, l'Ing. Roberto Pancaldi e l'Ing. Sergio Zannoli. Traduzione a cura di eds electronic data service - Bresso (Mi).

Tutti i diritti sono riservati. Stampato in Italia. Nessuna parte di questo libro può essere riprodotta, memorizzata in sistemi di archivio, o trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopia, registrazione o altri senza la preventiva autorizzazione scritta dell'editore.

Fotocomposizione: Corponove s.n.c. — Bergamo
Stampa: Tipo Lito Ferrari Cesare & C. — Clusone (BG)

RINGRAZIAMENTI

La realizzazione di un testo di programmazione è sempre difficile, ma lo è ancora di più se si vuole insegnare la programmazione elementare ed i concetti avanzati, comprendendo sia gli aspetti hardware che software. In questo caso è necessario realizzare un compromesso. L'autore desidera ringraziare per i numerosi suggerimenti costruttivi o variazioni: O. M. Barlow, Dennis L. Feick, Richard D. Reid, Stanley E. Ervin, Philip Hooper, Dennis B. Kitsz.
Un riconoscimento particolare va a Chris Williams per il suo contributo ai paragrafi sul set d'istruzioni e sulle strutture dati.
Qualsiasi suggerimento di miglioramento o variazione inviato all'autore sarà prezioso per le edizioni successive.

Diverse tabelle riportate al Capitolo Quattro riguardanti i codici esadecimali delle istruzioni dello Z80 sono state riprodotte su autorizzazione della Zilog Inc.

SOMMARIO

PREFAZIONE VIII

CAPITOLO 1 — CONCETTI FONDAMENTALI

Introduzione 1
Cos'è la programmazione? 1
Diagrammi di flusso 2
Rappresentazione delle informazioni 3

CAPITOLO 2 — ORGANIZZAZIONE HARDWARE DELLO Z80

Introduzione 23
Architettura del sistema 23
Organizzazione interna dello Z80 34
Formati delle istruzioni 36
Esecuzione delle istruzioni entro lo Z80 40
Sommario hardware 58

CAPITOLO 3 — TECNICHE FONDAMENTALI DI PROGRAMMAZIONE

Introduzione 59
Programmi aritmetici 60
Aritmetica BCD 68
Moltiplicazione 72
Divisione binaria 88
Sommario delle istruzioni 94
Le subroutine 95
Sommario 100

CAPITOLO 4 — IL SET DI ISTRUZIONI DELLO Z80

Introduzione 105
Classi di istruzioni 105
Sommario 129
Le istruzioni dello Z80: descrizione individuale 131

CAPITOLO 5 — TECNICHE DI INDIRIZZAMENTO

Introduzione 381
Modi possibili di indirizzamento 381
Modi di indirizzamento dello Z80 386
Impiego dei modi di indirizzamento dello Z80 389
Sommario 397

CAPITOLO 6 — TECNICHE DI INPUT/OUTPUT

Introduzione 399
Input Output 399
Trasferimento parallelo delle parole 404
Trasferimento seriale di bit 408
Sommario sulle periferiche 423
Scheduling d'Input Output 423
Sommario 437

CAPITOLO 7 — DISPOSITIVI D'INPUT/OUTPUT

Introduzione 439
Il PIO standard 439
Il registro di controllo interno 440
La programmazione di un PIO 443
Lo Z80 PIO della Zilog 443

CAPITOLO 8 — ESEMPI APPLICATIVI

Introduzione 447
Cancellazione di una sezione della memoria 447
Polling dei dispositivi di I/O 448
Accettazione di caratteri d'ingresso 448
Prova di un carattere 449
Prova parentesi 449
Generazione della parità 450
Conversione di codice: da ASCII a BCD 451
Conversione da esadecimale in ASCII 451
Ricerca dell'elemento più grande di una tabella 451
Somma di N elementi 452
Un calcolo di checksum 454
Conteggio di zeri 454
Trasferimento di blocchi 454
Trasferimento di blocchi BCD 455
Confronto di due numeri con segno a 16 bit 456
Bubble-sort 457
Sommario 462

CAPITOLO 9 — STRUTTURE DEI DATI

PARTE I - TEORIA
Introduzione 463
Puntatori 463
Liste 464
Ricerca e classificazione 468
Sommario del capitolo 469

PARTE II — ESEMPI DI PROGETTO
Introduzione 470
Rappresentazione dei dati per la lista 470
Una lista semplice 472
Lista alfabetica 479
Lista collegata 488
Sommario 497

CAPITOLO 10 — SVILUPPO DEL PROGRAMMA

Introduzione 499
Scelte di programmazione fondamentali 499
Il supporto software 501
Sequenza di sviluppo del programma 502
Alternative hardware 504
L'assemblatore 507
Assembly condizionale 514
Sommario 515

CAPITOLO 11 — CONCLUSIONE

Sviluppo tecnologico 517
La fase successiva 517

APPENDICE A 518
Tabella di conversione in esadecimale

APPENDICE B 519
Tabella di conversione ASCII

APPENDICE C 520
Tabella dei salti relativi

APPENDICE D 521
Conversione da decimale a BCD

APPENDICE E 522
Codici delle istruzioni dello Z80

APPENDICE F 529
Equivalenza tra Z80 ed 8080

APPENDICE G 530
Equivalenza tra 8080 e Z80

PREFAZIONE

Questo libro è stato progettato come un testo autonomo e completo per imparare la programmazione, usando lo Z80.

Può essere usato da una persona che non ha mai programmato prima, e dovrebbe essere di utilità a chiunque usi lo Z80.

Per la persona che ha già programmato, questo libro insegnerà le tecniche specifiche della programmazione usando (o lavorando attorno) le caratteristiche specifiche dello Z80.

Questo testo copre un arco di tecniche che vanno dalle elementari alle intermedie richieste per iniziare efficacemente la programmazione.

Questo testo mira a fornire un vero livello di competenza alla persona che desidera programmare usando questo microprocessore. Naturalmente, nessun libro insegnerà efficacemente come programmare, a meno che non si faccia veramente della pratica. Comunque, si spera che questo libro porterà il lettore al punto in cui egli sente di poter iniziare la programmazione da solo e poter risolvere problemi semplici o anche moderatamente complessi usando un microcomputer.

Questo libro è basato sull'esperienza dell'autore nell'insegnamento del come programmare microcomputer a più di 1.000 persone.

I capitoli vanno normalmente dal semplice al complesso. Per i lettori che hanno già imparato la programmazione elementare, il capitolo introduttivo può essere saltato. Per gli altri che non hanno mai programmato, le sezioni finali di alcuni capitoli possono richiedere una seconda lettura. Questo libro è stato progettato per portare sistematicamente il lettore attraverso i concetti fondamentali e le tecniche richieste per costruire programmi sempre più complessi. Inoltre, è fortemente consigliato di seguire l'ordine dei capitoli. Per risultati efficaci, è poi importante che il lettore tenti di risolvere quanti più esercizi possibile. Le difficoltà negli esercizi sono state attentamente graduate. Sono progettate per verificare che sia veramente capito quello che è stato presentato. Senza fare gli esercizi di programmazione, non sarà possibile capire il valore completo di questo libro come mezzo educativo. Diversi esercizi possono richiedere del tempo, come l'esercizio di moltiplicazione. Comunque, facendoli, voi veramente programmerete e imparerete facendoli. Ciò è indispensabile.

Per quelli che hanno acquisito un gusto per la programmazione alla fine di questo volume, è disponibile un volume simile: Il Libro delle Applicazioni dello Z80 (Z80 Applications Book).

Altri libri di questa serie coprono la programmazione per altri microprocessori molto ben conosciuti.

Per quelli che desiderano sviluppare la loro conoscenza dell'hardware si consiglia di consultare i libri: "Microprocessori" e "Tecniche di Interfacciamento dei Microprocessori".

Il contenuto di questo libro è stato controllato attentamente e crediamo sia affidabile. Comunque, è inevitabile che saranno trovati alcuni errori tipografici o altri.

L'autore sarà grato per qualsiasi commento da parte di attenti lettori in modo che le future edizioni possano beneficiare della loro esperienza. Sarà apprezzato qualsiasi altro consiglio per miglioramenti, come altri programmi desiderati, sviluppati o trovati interessanti da parte dei lettori.

CAPITOLO 1

CONCETTI FONDAMENTALI

INTRODUZIONE

Questo capitolo introdurrà i concetti fondamentali e le definizioni relative alla programmazione dei computer.

Il lettore già familiare con questi concetti potrebbe dare uno sguardo veloce al contenuto di questo capitolo e poi passare al capitolo 2. Comunque, è consigliato che anche i lettori con più esperienza, guardino il contenuto di questo capitolo introduttivo. Molti concetti significativi sonon presentati qui, incluso ad esempio, il complemento a 2, il sistema BCD ed altre rappresentazioni.

Alcuni di questi concetti possono essere nuovi al lettore; altri possono migliorare la conoscenza e la capacità di programmatori con esperienza.

COS'È LA PROGRAMMAZIONE?

Dato un problema si deve trovare una soluzione.

Questa soluzione, espressa come un procedimento graduale, è chiamata un *algoritmo*. Un algoritmo è una descrizione dettagliata graduale della soluzione ad un dato problema. Deve terminare in un numero finito di passi. Questo algoritmo può essere espresso in qualsiasi linguaggio o simbolismo. Un esempio semplice di un algoritmo è:

1 – inserite la chiave nella serratura
2 – girate la chiave per un giro completo a sinistra
3 – afferrate la maniglia della porta
4 – girate la maniglia della porta a sinistra e spingete la porta.

A questo punto, se l'algoritmo è corretto per il tipo di serratura implicato, la porta si aprirà. Questo procedimento a quattro passi si qualifica come un algoritmo per l'apertura della porta.

Una volta che la soluzione ad un problema è stata espressa nella forma di un algoritmo, l'algoritmo deve essere eseguito dal computer. Sfortunatamente, è ben noto il fatto che i computer non possono capire o eseguire un normale inglese parlato (o qualsiasi altra lingua umana). La ragione sta nella *ambiguità sintattica* di tutte le comuni lingue umane. Solo una "sottospecie" della lingua naturale può essere "capita" dal computer. Questa è chiamata *linguaggio di programmazione.*

Convertire un algoritmo in una sequenza di istruzioni di un linguaggio di programmazione è chiamato *programmazione*.

Per essere più specifici, la vera fase di traduzione dell'algoritmo nel linguaggio di programmazione è chiamata *codifica*.

La programmazione si riferisce veramente non solo alla codifica ma anche al progetto completo dei programmi e delle "strutture dei dati" i quali completeranno l'algoritmo.

La programmazione effettiva richiede non solo la comprensione delle possibili tecniche di esecuzione per gli algoritmi standard, ma anche un abile uso di tutte le risorse hardware dei computer, come i registri interni, la memoria e i dispositivi periferici, più un uso creativo delle appropriate strutture dei dati. Queste tecniche saranno coperte nei prossimi capitoli.

La programmazione richiede anche una stretta disciplina della documentazione, in modo che i programmi siano comprensibili agli altri, così come all'autore. La documentazione deve essere sia interna che esterna al programma.

La documentazione interna del programma si riferisce ai commenti posti nel corpo di un programma, per illustrarne il procedimento operativo.

La documentazione esterna si riferisce alle parti del progetto che sono separate dal programma: spiegazioni scritte, manuali e diagrammi di flusso.

DIAGRAMMI DI FLUSSO

Tra l'*algoritmo* e il *programma* è quasi sempre usata una fase intermedia. È chiamata *diagramma di flusso*. Un diagramma di flusso è semplicemente una rappresentazione simbolica dell'algoritmo espressa come una sequenza di rettangoli e di rombi contenenti i passi dell'algoritmo. I rettangoli sono usati per *comandi*, oppure per "statement (istruzioni) esecutivi".

I rombi sono usati per i test come: Se l'informazione X è vera, seguite la via indicata da A, oppure da B. Invece di presentare una definizione formale dei diagrammi di flusso a questo punto, introdurremo e tratteremo i diagrammi di flusso più tardi nel libro quando presenteremo i programmi.

Il diagramma di flusso è una fase intermedia altamente raccomandata tra la descrizione dettagliata dell'algoritmo e la vera codifica della soluzione. Eccezionalmente, è stato osservato

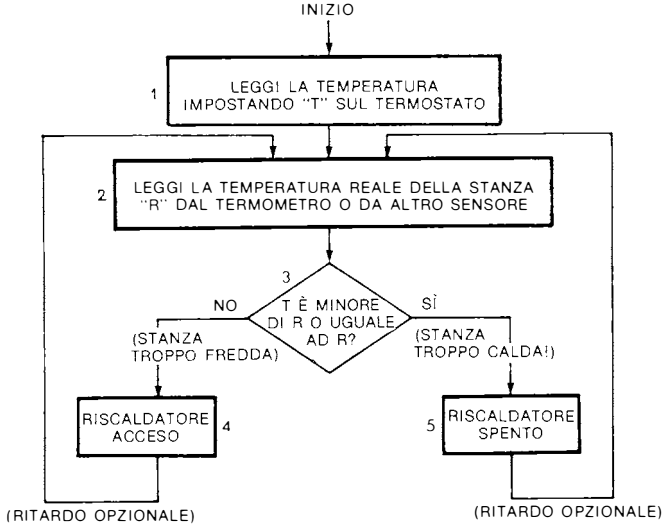


Figura 1-1: Diagramma di flusso per Mantere Costante la Temperatura della Stanza

che forse il 10% delle persone che programmano può scrivere un programma con buoni risultati senza dover fare un diagramma di flusso. Sfortunatamente, è stato anche osservato che il 90% della popolazione crede di appartenere a questo 10%! Il risultato: l'80% di questi programmi, in media, falliranno la prima volta che sono eseguiti su un computer. In breve, la maggior parte dei programmatori novizi raramente vedono la necessità di disegnare un diagramma di flusso. Questo risulta generalmente in programmi "non puliti" o errati. Devono allora passare molto tempo a provare ed a correggere il loro programma (questa è chiamata la fase di *debugging*, o di messa a punto).

La disciplina del diagramma di flusso è in tutti i casi altamente raccomandata.

Richiederà un piccolo ammontare di tempo addizionale prima della codifica, ma generalmente risulterà in un programma chiaro che si esegue correttamente e velocemente. Una volta che si è ben capito il modo di eseguire i diagrammi di flusso, una piccola percentuale di programmatori sarà capace di eseguire questa fase mentalmente senza doverla fare su carta. Sfortunatamente, in tali casi i programmi che essi scrivono saranno generalmente difficili da capire per qualsiasi altra persona senza la documentazione fornita dai diagrammi di flusso. Come risultato, è universalmente raccomandato che il diagramma di flusso sia usato come una ferrea disciplina per ogni programma significativo. Saranno forniti molti esempi nel corso del libro.

RAPPRESENTAZIONE DELLE INFORMAZIONI

Tutti i computer manipolano le informazioni nella forma di numeri o nella forma di caratteri. Esaminiamo qui le rappresentazioni interne e esterne delle informazioni in un computer.

RAPPRESENTAZIONI INTERNE DELLE INFORMAZIONI

Tutte le informazioni in un computer sono memorizzate come gruppi di bit. Un *bit* sta per *binary digit* ("0" oppure "1"). A causa delle limitazioni dell'elettronica convenzionale, l'unica rappresentazione pratica delle informazioni è una logica a due stati (la rappresentazione dello stato "0" e "1"). I due stati dei circuiti usati nell'elettronica digitale sono generalmente "on" o "off", e questi sono usati logicamente tramite i simboli "0" oppure "1". Siccome questi circuiti sono usati per compiere funzioni "logiche" i due stati sono chiamati "logica binaria". Come risultato virtualmente tutta la elaborazione delle informazioni è oggi eseguita in disposizione logica binaria. Nel caso dei microprocessori in generale, e dello Z80 in particolare, questi bit sono strutturati in gruppi di otto. Un gruppo di otto bit è chiamato *byte*. Un gruppo di quattro bit è chiamato *nibble*.

Esaminiamo ora come le informazioni sono internamente rappresentate in questo formato binario. Devono essere rappresentate due entità dentro il computer. La prima è il programma, che è una sequenza di istruzioni. La seconda sono i dati su cui opererà il programma, che potrebbero includere numeri o dati alfanumerici. Tratteremo qui tre rappresentazioni: programma, numeri e dati alfanumerici.

Rappresentazione del Programma

Tutte le istruzioni sono rappresentate internamente come byte singoli o multipli. Una cosiddetta "istruzione corta" è rappresentata da un byte singolo. Una istruzione più lunga sarà rappresentata da due o più byte. Siccome lo Z80 è un microprocessore ad otto bit, preleva byte successivamente dalla sua memoria. Perciò, una istruzione ad un solo byte ha sempre una capacità di esecuzione più veloce della istruzione a due o tre byte. Si vedrà più tardi che questa è una caratteristica importante del set di istruzioni di ogni microprocessore e in particolare dello Z80, dove è stato fatto uno sforzo speciale per fornire quanti più byte singoli possibile per migliorare la efficienza dell'esecuzione del programma. Comunque, la limitazione ad 8 bit in lunghezza è risultata in importanti restrizioni che saranno enucleate. Questo è un esempio classico del compromesso tra velocità e flessibilità nella programmazione. Il codice binario usato

per rappresentare le istruzioni è dettato dal fabbricante. Lo Z80, come ogni altro microprocessore, arriva equipaggiato di un set di istruzioni fisso. Queste istruzioni sono definite dal fabbricante e sono elencate alla fine di questo libro, con il loro codice. Ogni programma sarà espresso come una sequenza di queste istruzioni binarie. Le istruzioni dello Z80 sono presentate nel Capitolo 4.

Rappresentazione dei Dati Numerici

La rappresentazione dei numeri non è molto semplice, e devono essere distinti diversi casi. Per primo dobbiamo rappresentare i numeri interi, poi i numeri con il segno, cioè, i numeri positivi e negativi, e alla fine dobbiamo essere capaci di rappresentare i numeri decimali. Dedichiamoci ora a queste richieste e alle soluzioni possibili.

La rappresentazione dei numeri interi può essere compiuta usando una rappresentazione *binaria diretta.* La rappresentazione binaria diretta è semplicemente la rappresentazione del valore decimale di un numero nel sistema binario.

Nel sistema binario, il bit più a destra rappresenta 2 alla potenza 0. Il successivo alla sinistra rappresenta 2 alla potenza 1, il prossimo rappresenta 2 alla potenza 2 ed il bit più a sinistra rappresenta 2 alla potenza 7 = 128.

$$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

rappresenta

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

Le potenze di due sono:

$$2^7 = 128,\ 2^6 = 64,\ 2^5 = 32,\ 2^4 = 16,\ 2^3 = 8,\ 2^2 = 4,\ 2^1 = 2,\ 2^0 = 1$$

La rappresentazione binaria è analoga alla rappresentazione decimale dei numeri, dove "123" rappresenta:

$$\begin{array}{rcrcr} 1 & \times & 100 & = & 100 \\ +\ 2 & \times & 10 & = & 20 \\ +\ 3 & \times & 1 & = & 3 \\ \hline & & & & 123 \end{array}$$

Notate che $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.

In questa "notazione posizionale", ogni digit rappresenta una potenza di 10. Nel sistema binario, ogni digit binario o "bit" rappresenta una potenza di 2, invece di una potenza di 10 nel sistema decimale.

Esempio: "00001001" nel binario rappresenta:

$$\begin{array}{rcrcll} 1 & \times & 1 & = & 1 & (2^0) \\ 0 & \times & 2 & = & 0 & (2^1) \\ 0 & \times & 4 & = & 0 & (2^2) \\ 1 & \times & 8 & = & 8 & (2^3) \\ 0 & \times & 16 & = & 0 & (2^4) \\ 0 & \times & 32 & = & 0 & (2^5) \\ 0 & \times & 64 & = & 0 & (2^6) \\ 0 & \times & 128 & = & 0 & (2^7) \\ \hline \text{in decimale:} & & & = & 9 & \end{array}$$

Esaminiamo qualche altro esempio:

"10000001" rappresenta

$$
\begin{array}{rcrcr}
1 & \times & 1 & = & 1 \\
0 & \times & 2 & = & 0 \\
0 & \times & 4 & = & 0 \\
0 & \times & 8 & = & 0 \\
0 & \times & 16 & = & 0 \\
0 & \times & 32 & = & 0 \\
0 & \times & 64 & = & 0 \\
1 & \times & 128 & = & 128 \\
\hline
\text{in decimale:} & & & = & 129
\end{array}
$$

perciò "10000001" rappresenta il numero decimale 129.

Esaminando la rappresentazione binaria dei numeri, capirete perchè i bit sono numerati da 0 a 7, andando da destra a sinistra. Il bit 0 è "b_0" e corrisponde a 2^0. Il bit 1 è "b_1" e corrisponde a 2^1, e così continuando.

Decimale	Binario	Decimale	Binario
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010		
3	00000011		
4	00000100	.	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000		
9	00001001	.	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110		
15	00001111		
16	00010000		
17	00010001		
.		254	11111110
31	00011111	255	11111111

Figura 1-2: Tabella Decimale-Binario

Gli equivalenti binari dei numeri da 0 a 255 sono mostrati nella Figura 1-2.

Esercizio 1-1: *Qual è il valore decimale di "11111100"?*

Conversione Decimale - Binario

Calcoliamo viceversa l'equivalente binario di "11" decimale:

11 ÷ 2 = 5	rimane 1	→ 1		(LSB)
5 ÷ 2 = 2	rimane 1	→ 1		
2 ÷ 2 = 1	rimane 0	→ 0		
1 ÷ 2 = 0	rimane 1	→ 1		(MSB)

L'equivalente binario è 1011 (leggete la colonna più a destra dal basso all'alto).
L'equivalente binario di un numero decimale può essere ottenuto dividendo successivamente per 2 fino a quando è ottenuto un quoziente 0.

Esercizio 1-2: *Qual è il binario per 257?*

Esercizio 1-3: *Convertite 19 in binario, e poi nuovamente in decimale.*

Operazioni con Dati Binari

Le regole aritmetiche per i numeri binari sono semplici.
Le regole per l'addizione sono:

$$\begin{array}{rcl} 0 + 0 & = & 0 \\ 0 + 1 & = & 1 \\ 1 + 0 & = & 1 \\ 1 + 1 & = & (1)\ 0 \end{array}$$

dove (1) denota un "riporto" di 1 (notate che "10" è l'equivalente binario di "2" decimale). La sottrazione binaria sarà eseguita "addizionando il complemento" e sarà spiegata una volta che impariamo come rappresentare i numeri negativi.

Esempio:

$$\begin{array}{r} (2) \\ +(1) \\ \hline =(3) \end{array} \qquad \begin{array}{r} 10 \\ +01 \\ \hline 11 \end{array}$$

L'addizione è eseguita proprio come nel decimale, sommando le colonne, da destra a sinistra:
Addizionando la colonna più a destra:

$$\begin{array}{r} 10 \\ +01 \\ \hline \end{array}$$

(0 + 1 = 1. Nessun riporto).

Addizionando la prossima colonna:

$$\begin{array}{r} 10 \\ +01 \\ \hline 11 \end{array} \quad (1 + 0 = 1.\ \text{Nessun riporto}).$$

Esercizio 1-4: *Calcolate 5 + 10 in binario. Verificate che il risultato sia 15.*

Qualche esempio addizionale di addizione binaria:

```
  0010     (2)
+ 0001     (1)
------
= 0011     (3)

  0011     (3)
+ 0001     (1)
------
= 0100     (4)
```

Questo ultimo esempio illustra il ruolo del riporto.
Guardando ai bit più a destra: 1 + 1 = (1) 0
È generato un riporto di 1, che deve essere addizionato ai bit successivi:

```
  001  — La colonna 0 è già stata addizionata
+ 000  —
+   1    (riporto)
------
= (1) 0 — dove (1) indica un nuovo riporto
          nella colonna 2
```

Il risultato finale è: 0100.

Un altro esempio:

```
  0111      ( 7)
+ 0011    + ( 3)
------
  1010    =  10
```

In questo esempio, viene generato di nuovo un riporto, fino alla colonna più a sinistra.

Esercizio 1-5: *Calcolate il risultato di:*

```
  1111
+ 0001
------
= ?
```

Il risultato è limitato a quattro bit?

Con otto bit, è perciò possibile presentare direttamente i numeri da "00000000" a "11111111", cioè, da "0" a "255". Due ostacoli dovrebbero essere immediatamente visibili. Primo, stiamo rappresentando solo numeri positivi. Secondo, la grandezza di questi numeri è limitata a 255 se usiamo solo otto bit. Consideriamo separatamente ognuno di questi problemi.

Binario con Segno

Nella rappresentazione in binario con il segno, il bit più a sinistra è usato per indicare il segno del numero.

Tradizionalmente, "0" è usato per denotare un numero *positivo* mentre "1" per denotare un numero *negativo.* Allora "11111111" rappresenterà −127, mentre "01111111" rappresenterà +127.

Ora possiamo presentare numeri positivi e negativi, ma abbiamo ridotto la grandezza massima di questi numeri a 127.

Esempio: "0000 0001" rappresenta +1 (il primo "0" è "+", seguito da "000 0001" = 1). "1000 0001" è −1 (il primo "1" è "−").

Esercizio 1-6: *Qual è la rappresentazione di "−5" nel binario con segno?*

Dedichiamoci ora al problema della grandezza: per rappresentare numeri più grandi, sarà necessario usare un numero più grande di bit. Per esempio, se usiamo sedici bit (due byte) per rappresentare i numeri, saremo in grado di rappresentare i numeri da – 32K a + 32K in binario con segno (1K nel gergo dei computer rappresenta 1024). Il bit 15 è usato per il segno, ed i rimanenti 15 bit (dal bit 14 al bit 0) sono usati per la grandezza: $2^{15} = 32K$. Se questa grandezza è ancora troppo piccola, useremo 3 byte o più. Se desideriamo rappresentare grandi numeri interi, sarà necessario usare un più grande numero di byte internamente per rappresentarli. Ecco perché la maggior parte dei BASIC semplici, ed altri linguaggi, forniscono solo una precisione limitata per i numeri interi. In questo modo, possono usare un formato interno più breve per i numeri che manipolano. Versioni migliori del BASIC, oppure di questi altri linguaggi, forniscono un numero più grande di cifre decimali significative al prezzo di un grande numero di byte per ogni numero.

Ora, risolviamo un altro problema, quello della efficienza in termini di velocità. Tenteremo di eseguire una addizione con la rappresentazione binaria con segno che abbiamo introdotto. Addizioniamo "–5" e "+7".

+7 è rappresentato da	00000111
–5 è rappresentato da	10000101
La somma binaria è:	10001100, oppure –12

Questo non è il risultato corretto. Il risultato corretto dovrebbe essere +2. Per usare questa rappresentazione, devono essere fatte alcune operazioni speciali, a seconda del segno. Ciò si risolve in un'aumentata complessità e in una ridotta prestazione. In altre parole, l'addizione binaria dei numeri con segno non "funziona correttamente". Ciò è seccante. Chiaramente, il computer non solo rappresenta informazioni, ma esegue anche l'aritmetica su di esse.

La soluzione a questo problema è chiamata la rappresentazione in *complemento a due*, che sarà usata invece della rappresentazione *binaria con segno*. Per introdurre il complemento a due, occorre prima introdurre una fase intermedia: *il complemento ad uno.*

Il complemento ad uno

Nella rappresentazione in complemento a uno, tutti i numeri interi positivi sono rappresentati nel loro corretto formato binario. Per esempio "+3" è rappresentato come sempre da 00000011. Comunque, il suo complemento "–3" è ottenuto complementando (facendo il complemento) ogni bit nella rappresentazione originale. Ogni 0 è trasformato in un 1 e ogni 1 è trasformato in uno 0. Nel nostro esempio, la rappresentazione in complemento ad uno di "–3" sarà 11111100.

Un altro esempio:

+ 2 è 00000010
– 2 è 11111101

Notate che, in questa rappresentazione, i numeri positivi iniziano con uno "0" alla sinistra, e quelli negativi con "1" alla sinistra.

Esercizio 1-7: *La rappresentazione di "+6" è "00000110". Qual è la rappresentazione di "–6" in complemento ad uno?*

Come prova, sommiamo meno 4 e più 6:

	–4 è 11111011
	+6 è 00000110
La somma è	(1) 00000001 dove (1) indica un riporto

Il "risultato corretto" dovrebbe essere "2", ovvero "00000010".

Proviamo di nuovo:

	−3 è	11111100
	−2 è	11111101
La somma è	(1)	00000001

oppure "1", più un riporto. Il risultato corretto dovrebbe essere "−5". La rappresentazione di "−5" è 11111010. Non ha funzionato.

Questa rappresentazione rappresenta numeri positivi e negativi.

Comunque, il risultato di una normale addizione non uscirà sempre "correttamente". Noi useremo ancora un'altra rappresentazione. Si sviluppa dal complemento ad uno ed è chiamata la rappresentazione in complemento a due.

La rappresentazione in complemento a due

Nella rappresentazione in complemento a due, i numeri positivi sono ancora rappresentati, come al solito, in binario con segno, proprio come in complemento ad uno. La differenza sta nella rappresentazione dei numeri negativi. Un numero negativo rappresentato in complemento a due è ottenuto calcolando per prima il complemento a uno, e poi *aggiungendo uno.*

Esaminiamolo in un esempio:

+3 è rappresentato nel binario con segno da 00000011.
La sua rappresentazione in complemento ad uno è 11111100.
Il complemento a due è ottenuto aggiungendo 1. È 11111101.
Proviamo una addizione:

(3)	00000011
+ (5)	+ 00000101
= (8)	= 00001000

Il risultato è corretto.

Proviamo una sottrazione:

(3)	00000011
(−5)	+ 11111011
	= 11111110

Identifichiamo il risultato calcolando il complemento a due:

il complemento ad uno di 11111110 è	00000001
aggiungendo 1	+ 1
perciò il complemento a due è	00000010 oppure +2

Il nostro risultato "11111110" rappresenta "−2". È corretto.

Abbiamo provato adesso l'addizione e la sottrazione, ed i risultati erano corretti (ignorando il riporto). Sembra che il complemento a 2 funzioni!

Esercizio 1-8: *Qual è la rappresentazione in complemento a due di "+127"?*

Esercizio 1-9: *Qual è la rappresentazione in complemento a due di "−128"?*

Ora sommiamo +4 e −3 (la sottrazione è eseguita sommando il complemento a due):

	+4 è	00000100
	−3 è	11111101
Il risultato è	(1)	00000001

Se ignoriamo il riporto, il risultato è 00000001, cioè, "1" in decimale. Questo è il risultato corretto. Senza dare la prova matematica completa, affermiamo semplicemente che questa rappresentazione funziona. Nel complemento a due, è possibile aggiungere o sottrarre numeri con segno senza badare al segno. Usando le regole usuali dell'addizione binaria, il risultato esce correttamente, incluso il segno. Il riporto è ignorato. Questo è un vantaggio molto significativo. In caso contrario si dovrebbe correggere il risultato relativamente al segno ogni volta, provocando un tempo di addizione o di sottrazione molto più lungo.

Per completezza, diciamo che il complemento a due è semplicemente la rappresentazione più conveniente da usare per i processori più semplici come i microprocessori. Su processori complessi, potrebbero essere usate altre rappresentazioni. Per esempio, potrebbe essere usato il complemento a uno, ma richiede collegamenti elettrici speciali per "correggere il risultato".

Da questo momento, tutti i numeri interi con segno saranno rappresentati implicitamente all'interno del processore nella notazione in complemento a due. La Figura 1-3 riporta una tabella dei numeri in complemento a due.

Esercizio 1-10: *Quali sono i numeri più grandi e più piccoli che si possono rappresentare nella notazione in complemento a due, usando solo un byte?*

Esercizio 1-11: *Calcolate il complemento a due di 20. Poi calcolate il complemento a due del vostro risultato. Trovate di nuovo 20?*

I seguenti esempi serviranno a dimostrare le regole del complemento a due. In particolare, C denota una condizione di riporto (o prestito) possibile (È il bit 8 del risultato). V denota una eccedenza di capacità (overflow) del complemento a due, cioè, quando il segno del risultato è variato "accidentalmente" perchè i numeri sono troppo grandi. È un riporto essenzialmente interno dal bit 6 nel bit 7 (il bit del segno). Questo sarà chiarito di seguito.

Dimostriamo ora il ruolo del riporto "C" e l'eccedenza di capacità "V" (overflow "V").

Il riporto C

Ecco un esempio di un riporto:

(128)		10000000
+ (129)		+ 10000001
(257) =	(1)	00000001

dove (1) indica un riporto.

Il risultato richiede un nono bit (bit "8", poichè il bit più a destra è "0"). È il bit di riporto. Se presumiamo che il riporto è il nono bit del risultato, riconosciamo che il risultato sia 100000001 = 257.

Comunque, il riporto deve essere riconosciuto e trattato con attenzione. All'interno del microprocessore, i registri usati per contenere le informazioni hanno generalmente una larghezza di otto bit. Quando si memorizza il risultato, saranno contenuti solo i bit da 0 a 7.

Perciò un riporto richiede sempre un'azione speciale: deve essere rivelata tramite istruzioni speciali, e poi elaborata.

L'elaborazione del riporto significa memorizzarlo da qualche parte (con una istruzione speciale), oppure ignorarlo, oppure decidere che è un errore (se il più grande risultato consentito è "11111111").

+	Codice in complemento a 2	−	Codice in complemento a 2
+127	01111111	−128	10000000
+126	01111110	−127	10000001
+125	01111101	−126	10000010
		−125	10000011
		...	
+65	01000001	−65	10111111
+64	01000000	−64	11000000
+63	00111111	−63	11000001
...		...	
+33	00100001	−33	11011111
+32	00100000	−32	11100000
+31	00011111	−31	11100001
...		...	
+17	00010001	−17	11101111
+16	00010000	−16	11110000
+15	00001111	−15	11110001
+14	00001110	−14	11110010
+13	00001101	−13	11110011
+12	00001100	−12	11110100
+11	00001011	−11	11110101
+10	00001010	−10	11110110
+9	00001001	−9	11110111
+8	00001000	−8	11111000
+7	00000111	−7	11111001
+6	00000110	−6	11111010
+5	00000101	−5	11111011
+4	00000100	−4	11111100
+3	00000011	−3	11111101
+2	00000010	−2	11111110
+1	00000001	−1	11111111
+0	00000000		

Figura 1-3: Tabella del Codice in Complemento a 2

Eccedenza di capacità V (overflow V)

Ecco un esempio di overflow:

bit 6
bit 7

```
  01000000        (64)
+ 01000001      + (65)
 ----------------------
= 10000001    = (−127)
```

È stato generato un riporto interno dal bit 6 nel bit 7. Questo è chiamato overflow.
Il risultato è ora negativo, "per caso". Questa situazione deve essere rivelata, in modo che possa venire corretta.

Esaminiamo un'altra situazione:

	11111111	(−1)
	+11111111	+(−1)
=(1)	11111110	=(−2)
▼ riporto (carry)		

In questo caso, è stato generato un riporto interno dal bit 6 nel bit 7, e anche dal bit 7 nel bit 8 (il riporto C formale che abbiamo esaminato nella sezione precedente). Le regole dell'aritmetica del complemento a due specificano che questo riporto dovrebbe essere ignorato. Il risultato è allora corretto.

Questo perchè il riporto dal bit 6 nel bit 7 non ha cambiato il bit del segno.

Questa è una condizione di *overflow*. Nel caso di numeri negativi l'overflow non è semplicemente un riporto dal bit 6 al bit 7. Esaminiamo un altro esempio:

	11000000	(−64)
	+10111111	(−65)
=(1)	01111111	(+127)
▼ riporto (carry)		

Questa volta non c'è un riporto interno dal bit 6 al bit 7, ma un riporto esterno. Il risultato non è corretto in quanto è stato cambiato il bit 7. È quindi necessario segnalare una condizione di overflow.

L'overflow avverrà in quattro situazioni:

1 – aggiungendo grandi numeri positivi.
2 – aggiungendo grandi numeri negativi.
3 – sottraendo un grande numero positivo da un grande numero negativo.
4 – sottraendo un grande numero negativo da un grande numero positivo.

Ora, miglioriamo la nostra definizione dell'overflow:

Tecnicamente, l'indicatore di overflow, un bit speciale riservato per questo scopo, e chiamato "flag" (bandiera), sarà collocato quando c'è un riporto dal bit 6 nel bit 7 e nessun riporto esterno, oppure quando non c'è nessun riporto dal bit 6 nel bit 7 ma c'è un riporto esterno. Questa indica che il bit 7, cioè il segno del risultato, è stato accidentalmente cambiato. Per il lettore più tecnico, il flag overflow si ottiene facendo l'OR Esclusivo del valore d'ingresso e del valore d'uscita del bit 7 (il bit del segno). Praticamente ogni microprocessore è fornito di uno speciale flag di overflow per rivelare automaticamente questa azione, la quale richiede una azione correttiva.

L'overflow indica che il risultato di una addizione o di una sottrazione richiede più bit di quanto siano disponibili nel registro standard a otto bit usato per contere il risultato.

Il Riporto e l'Overflow

Il bit di riporto e overflow sono chiamati "flag".

Sono contenuti in ogni microprocessore, e nel prossimo capitolo impareremo ad usarle per una effettiva programmazione.

Questi due indicatori sono disposti in un registro speciale chiamato il registro dei flag o di "stato".

Questo registro contiene anche degli indicatori addizionali le cui funzioni saranno chiarite nel Capitolo 4.

Esempi

Ora illustriamo il funzionamento del riporto e dell'overflow in esempi veri. In ogni esempio, il

simbolo V denota l'overflow, e C il riporto.

Se non c'è stato nessun overflow, V = 0. Se c'è stato un overflow, V = 1 (lo stesso per il riporto). Ricordatevi che le regole del complemento a due specificano che il riporto sia ignorato. (La prova matematica non è fornita qui).

Positivo-Positivo

```
  00000110        (+ 6)
+ 00001000        (+ 8)
-----------------------
  00001110        (+14)     V:0      C:0
(CORRETTO)
```

Positivo-Positivo con Overflow

```
  01111111        (+127)
+ 00000001        (+  1)
-----------------------
  10000000        (-128)    V:1      C:0
Questo non è valido perchè è avvenuto un overflow
(ERRORE)
```

Positivo-Negativo (risultato positivo)

```
    00000100      (+4)
    11111110      (-2)
-----------------------
  (1) 00000010    (+2)      V:0      C:1 (ignorato)
(CORRETTO)
```

Positivo-Negativo (risultato negativo)

```
  00000010        (+2)
+ 11111100        (-4)
-----------------------
  11111110        (-2)      V:0      C:0
(CORRETTO)
```

Negativo-Negativo

```
    11111110      (-2)
+   11111010      (-4)
-----------------------
  (1) 11111010    (-6)      V:0      C:1 (ignorato)
(CORRETTO)
```

Negativo-Negativo con Overflow

```
    10000001      (-127)
+   11000010      (-  62)
-----------------------
  (1) 01000011    (   67)   V:1      C:1
(ERRORE)
```

Questa volta si è verificato un underflow, sommando due grandi numeri negativi. Il risultato sarebbe −189, che è troppo grande per risiedere in otto bit.

Esercizio 1-12: *Completate le seguenti addizioni. Indicate il risultato, il riporto C, l'overflow V, e se il risultato è corretto o no:*

```
  10111111      (__)                11111010      (__)
+ 11000001      (__)              + 11111001      (__)
-----------     V:__  C:__        -----------     V:__  C:__
☐CORRETTO       ☐ERRORE           ☐CORRETTO       ☐ERRORE
```

```
 00010000     (__)                    01111110     (__)
+01000000     (__)                   +00101010     (__)
 --------     V: __  C: __            --------     V: __  C: __
 CORRETTO       ERRORE                CORRETTO       ERRORE
```

Esercizio 1-13: *Potete mostrare un esempio di overflow quando sommate un numero positivo e uno negativo? Perchè?*

Rappresentazioni a Formato Fisso

Ora sappiamo come rappresentare i numeri interi con segno. Comunque, non abbiamo ancora risolto il problema della grandezza. Se vogliamo rappresentare numeri interi più grandi, avremo bisogno di diversi byte. Per eseguire in modo efficiente le operazioni aritmetiche, è necessario usare un numero fisso di byte piuttosto che uno variabile. Perciò, una volta che è scelto il numero dei byte, è fissata la grandezza massima del numero che può essere rappresentato.

Esercizio 1-14: *Quali sono i numeri più grandi e più piccoli che possono essere rappresentati in due byte usando il complemento a due?*

Il problema della grandezza

Nella somma di numeri ci siamo limitati ad otto bit perchè il processore che useremo opera interamente su otto bit alla volta. Comunque, questo ci limita ai numeri nel range da −128 a +127.

Chiaramente, questo non è sufficiente per molte applicazioni.

Sarà usata la precisione multipla per aumentare il numero dei digit che possono essere rappresentati. Può essere allora usato un formato a due byte, tre byte, oppure N byte. Per esempio, esaminiamo un formato in "doppia precisione" su 16 bit:

```
00000000    00000000    è  "0"
00000000    00000001    è  "1"

...
01111111    11111111    è  "32767"
11111111    11111111    è  "−1"
11111111    11111110    è  "−2"
```

Esercizio 1-15: *Qual è il più grande numero intero negativo che può essere rappresentato con un formato a precisione tripla in complemento a due?*

Comunque, questo metodo risulterà svantaggioso. Quando si sommano due numeri, per esempio, dovremo generalmente sommarli otto bit alla volta. Questo sarà spiegato nel Capitolo 3 (Tecniche Fondamentali di Programmazione). Risulta quindi una elaborazione più lenta. Poi, questa rappresentazione usa 16 bit per ogni numero, anche se potrebbe essere rappresentato con soli otto bit. È perciò comune usare 16 o forse 32 bit, ma solo raramente di più.

Consideriamo il seguente punto importante: qualunque sia il numero scelto di bit N per la rappresentazione del complemento a due, esso è fissato. Alcuni bit saranno perduti, se ogni risultato o calcolo intermedio dovesse generare un numero che richiede più di N bit. Normalmente, il programma conserva gli N bit più a sinistra (i più significativi) e lascia cadere quelli di basso ordine. Questo è chiamato troncamento del risultato.

Ecco un esempio nel sistema decimale, che usa una rappresentazione a sei cifre:

(Nota: Secondo la notazione inglese si usa il punto decimale al posto della virgola).

```
      123456
x        1.2
----------
      246912
     123456
----------
  - 148147.2
```

Il risultato richiede 7 cifre! Il "2" dopo il punto decimale sarà lasciato cadere e il risultato finale sarà 148147. È stato troncato. Generalmente, fino a quando non è perduta la posizione del punto decimale, questo metodo è usato per estendere la gamma delle operazioni che possono essere eseguite, a spese della precisione.

Il problema è lo stesso nel binario. I dettagli di una moltiplicazione binaria saranno mostrati nel Capitolo 4.

Questa rappresentazione a formato fissato può causare una perdita di precisione, ma può essere sufficiente per i calcoli usuali o per le operazioni matematiche.

Sfortunatamente, nel caso della contabilità non è tollerabile nessuna perdita di precisione. Deve essere usata un'altra rappresentazione dovunque sia essenziale la precisione nel risultato. La soluzione normalmente usata è il BCD, o decimale codificato binario.

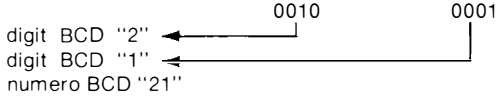
Rappresentazione BCD

Il principio usato per rappresentare i numeri nel BCD è di codificare separatamente ogni digit decimale, e di usare tanti bit quanti sono necessari per rappresentare esattamente il numero, completo. Sono necessari quattro bit per codificare ogni cifra da 0 a 9. Tre bit fornirebbero soltanto otto combinazioni, e perciò non possono codificare le dieci cifre. Quattro bit permettono sedici combinazioni e sono perciò sufficienti per codificare le cifre "0" fino a "9". Può anche essere annotato che sei dei possibili codici non saranno usati nella rappresentazione BCD (vedere la Figura 1-3). Questo risulterà più avanti in un problema potenziale durante addizioni e sottrazioni che dovremo risolvere. Dal momento in cui sono necessari solo 4 bit per codificare un digit BCD, possono essere codificati due digit BCD in ogni byte. Questo è chiamato *"BCD compattato"*

CODICE	SIMBOLO BCD	CODICE	SIMBOLO BCD
0000	0	1000	8
0001	1	1001	9
0010	2	1010	non utilizzato
0011	3	1011	non utilizzato
0100	4	1100	non utilizzato
0101	5	1101	non utilizzato
0110	6	1110	non utilizzato
0111	7	1111	non utilizzato

Figura 1-4: Tabella del codice BCD

Come esempio, "00000000" sarà "00" nel BCD. "1001 1001" sarà "99".
Un codice BCD è letto come segue:

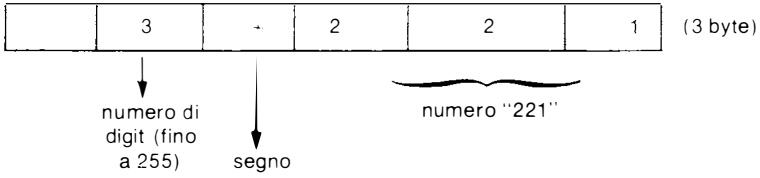


Esercizio 1-16: *Qual è la rappresentazione BCD per "29"? "91"?*

Esercizio 1-17: *"10100000" è una rappresentazione BCD valida? Perchè?*

Saranno usati tanti byte quanti sono necessari per rappresentare tutti i digit BCD. Tipicamente saranno usati uno o più nibble all'inizio della rappresentazione per indicare il numero totale di nibble, cioè, il numero totale di digit BCD usati. Sarà usato un altro nibble o byte per denotare la posizione del punto decimale. Comunque, le convenzioni possono variare.

Ecco un esempio di una rappresentazione per numeri interi BCD a più byte:



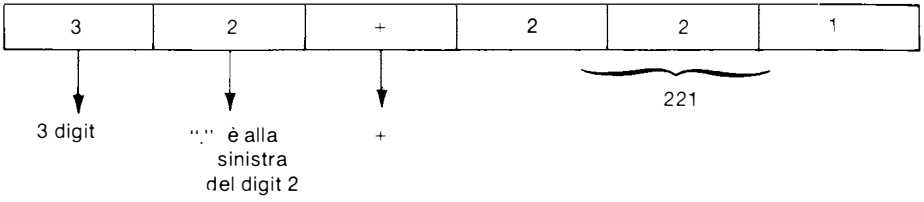
Questo rappresenta + 221.
(Per esempio il segno può essere rappresentato da 0000 per +, e 0001 per −)

Esercizio 1-18: *Usando la stessa convenzione, rappresentate "− 23123". Mostratelo nel formato BCD, come sopra, e poi nel binario.*

Esercizio 1-19: *Mostrate il BCD per "222" e "111", poi per il risultato di 222 × 111. (Calcolate il risultato a mano, poi mostratelo nella rappresentazione precedente).*

La rappresentazione BCD si presta facilmente per i numeri decimali.

Per esempio, + 2.21 può essere rappresentato da:



Il vantaggio del BCD è che produce risultati assolutamente corretti. Il suo svantaggio è che usa una grande quantità di memoria e si risolve in operazioni matematiche lente.

Questo è accettabile nelle condizioni di contabilità e non è normalmente usato in altri casi.

Esercizio 1-20: *Quanti bit sono richiesti per codificare "9999" in BCD? Ed in complemento a due?*

Abbiamo ora risolto i problemi associati con la rappresentazione dei numeri interi, numeri interi con segno e anche dei grandi numeri interi. Abbiamo anche già presentato un metodo possibile per rappresentare i numeri decimali, con la rappresentazione BCD. Ora esaminiamo il problema della rappresentazione dei numeri decimali in un formato a lunghezza fissa.

Rappresentazione a Virgola Mobile (Floating Point)

Il principio fondamentale è quello che i numeri decimali devono essere rappresentati con un formato fisso. Per non sprecare bit, la rappresentazione *normalizzerà* tutti i numeri.

Per esempio, "0.000123" spreca tre zeri alla sinistra del numero, che non ha nessun signifi-

cato eccetto quello di indicare la posizione del punto decimale. La normalizzazione di questo genera $.123 \times 10^{-3}$. ".123" è chiamato *mantissa normalizzata.* "−3" è chiamato *esponente.* Abbiamo normalizzato questo numero eliminando tutti gli zeri senza significato alla sinistra e regolando l'esponente.

Consideriamo un altro esempio:

22.1 è normalizzato come $.221 \times 10^2$
oppure $M \times 10^E$ dove M è la mantissa, ed E è l'esponente.

Si può subito notare che un numero normalizzato è caratterizzato da una mantissa inferiore a 1 e più grande o uguale a .1 in tutti i casi dove il numero non è zero. In altre parole, questo può essere rappresentato matematicamente da:

$$.1 \leq M < 1 \text{ oppure } 10^{-1} \leq M < 10^0$$

in modo simile, nella rappresentazione binaria:

$$2^{-1} \leq M < 2^0 \text{ (oppure } .5 \leq M < 1)$$

Dove M è il valore assoluto della mantissa (ignorando il segno).

Per esempio:

$$111.01 \text{ è normalizzato come: } .11101 \times 2^3$$

La mantissa è .11101.
L'esponente è 3.

Ora che abbiamo definito il principio della rappresentazione, esaminiamo il formato vero. Una tipica rappresentazione a virgola mobile appare di seguito.

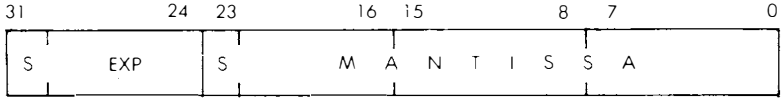


Figura 1-5: Tipica Rappresentazione a Virgola mobile

Nella rappresentazione usata in questo esempio, vengono impiegati quattro byte per un totale di 32 bit.

Il primo byte alla sinistra dell'illustrazione viene impiegato per rappresentare l'esponente. Sia l'esponente che la mantissa saranno presentati in complemento a due. Come risultato, l'esponente massimo sarà −128. "S" nella Figura 1-5 denota il bit del segno.

Sono usati tre byte per rappresentare la mantissa. Poichè il primo bit nella rappresentazione in complemento a due indica il segno, questo lascia 23 bit per la rappresentazione della grandezza della mantissa.

Esercizio 1-21: *Quanti digit decimali può rappresentare la mantissa con i 23 bit?*

Questo è solo un esempio della rappresentazione a virgola mobile. È possibile usare solo tre byte, oppure è possibile usarne di più. La rappresentazione a quattro byte proposta sopra è soltanto una rappresentazione comune che rappresenta un ragionevole compromesso in termini di precisione, grandezza dei numeri, utilizzazione della memoria, e efficienza nella operazione matematica.

Abbiamo ora esplorato i problemi associati con la rappresentazione dei numeri e sappiamo come presentarli nella forma dei numeri interi, con un segno, o nella forma decimale.

Adesso esaminiamo come rappresentare internamente i dati alfanumerici.

Rappresentazione dei Dati Alfanumerici

La rappresentazione dei dati alfanumerici, cioè i caratteri, è molto semplice: tutti i caratteri sono codificati in un codice ad otto bit. Solo due codici sono di uso generale nel mondo del computer, il Codice ASCII ed il Codice EBCDIC. ASCII sta per "American Standard Code for Information Interchange" (Codice Standard Americano per lo Scambio di Informazioni) ed è universalmente usato nel mondo dei microprocessori. EBCDIC è una variazione dell'ASCII usato dall'IBM, e perciò non usato nel mondo dei microcomputer a meno che non si realizzi l'interfaccia ad un terminale IBM.

Esaminiamo brevemente la codifica ASCII. Dobbiamo codificare 26 lettere dell'alfabeto sia per le maiuscole che le minuscole, più 10 simboli numerici, più, forse 20, simboli speciali addizionali. Questo può essere semplicemente compiuto con 7 bit, che permettono 128 codici possibili. (vedere Figura 1-6). Perciò, tutti i caratteri sono codificati in 7 bit. L'ottavo bit, quando è usato, è il *bit di parità*. La parità è una tecnica per verificare che i contenuti di un byte non siano stati accidentalmente cambiati. Viene contato il numero degli 1 nel byte e l'ottavo bit è posto ad uno se il conteggio era dispari, rendendo perciò pari il totale. Questa è chiamata la parità pari. Si può anche usare la parità dispari, cioè, scrivendo l'ottavo bit (quello più a sinistra) in modo tale che il numero totale degli 1 nel byte sia dispari. Esempio: calcoliamo il bit di parità per "0010011" usando la parità pari. Il numero degli 1 è 3. Il bit di parità deve perciò essere un 1 in modo che il numero totale di bit sia 4, cioè, pari. Il risultato è 10010011, dove il primo 1 è il bit di parità e 0010011 identifica il carattere.

La tabella dei codici ASCII a 7 bit è mostrata nella Figura 1-6. In pratica, è usata "come è", cioè senza parità, aggiungendo uno 0 nella posizione più a sinistra, oppure con la parità, aggiungendo un appropriato bit extra alla sinistra.

Esercizio 1-22: *Calcolate la rappresentazione ad 8 bit delle cifre da "0" a "9", usando la parità pari. (Questo codice sarà usato negli esempi applicativi del capitolo 8).*

Esercizio 1-23: *Lo stesso per le lettere da "A" ad "F".*

ESAD.	MSD	0	1	2	3	4	5	6	7
LSD	BIT	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPAZIO	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Figura 1-6: Tabella di Conversione ASCII (Per le abbreviazioni vedere l'Appendice B)

In situazioni specializzate come quelle delle comunicazioni, possono essere usate altre codificazioni come il codice a correzione di errore. Comunque questi esulano dallo scopo di questo libro.

Abbiamo esaminato le rappresentazioni usuali sia per il programma che per i dati all'interno del computer. Esaminiamo ora le possibili rappresentazioni esterne.

RAPPRESENTAZIONE ESTERNA DELL'INFORMAZIONE

La rappresentazione esterna si riferisce al modo con cui è presentata l'informazione a chi la usa, cioè, generalmente un programmatore. L'informazione può essere presentata esternamente in tre formati: binaria, ottale o esadecimale e simbolica.

1. Binario

È stato visto che l'informazione è memorizzata internamente in *byte*, che sono una sequenza di otto bit (0 oppure 1). A volte è desiderabile mostrare questa informazione interna direttamente nel suo formato binario e quella è chiamata *rappresentazione binaria*. Un esempio semplice è fornito dai Diodi Emettitori di Luce (LED) che sono essenzialmente lampade in miniatura, sul pannello frontale del microcomputer. In questo caso, per un microprocessore ad otto bit, un pannello frontale sarà tipicamente attrezzato di otto LED per mostrare il contenuto di qualsiasi registro interno. (Un registro è usato per contenere otto bit di informazione e sarà descritto nel Capitolo 2). Un LED acceso indica un uno. Uno zero è indicato da un LED spento. Tale rappresentazione binaria può essere usata per una buona messa a punto (debugging) di un programma complesso, specialmente se implica input·output (ingresso uscita), ma è naturalmente non pratica a livello umano. Ciò perchè, nella maggior parte dei casi, è più comodo osservare l'informazione nella forma simbolica. Così "9" è molto più facile da capire o ricordare di "1001". Sono state inventate rappresentazioni più comode, che migliorano l'interfaccia uomo - macchina.

2. Ottale ed Esadecimale

"Ottale" ed "Esadecimale" codificano rispettivamente tre o quattro bit binari in un simbolo unico. Nel sistema ottale, ogni combinazione di tre bit binari è rappresentata da un numero tra 0 e 7. L'"ottale" è un formato che usa tre bit, dove ogni combinazione di tre bit è rappresentata da un simbolo tra 0 e 7:

binario	ottale
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figura 1-7: Simboli Ottali

Per esempio, il binario "00 100 100" è rappresentato da:

0 4 4

ovvero "044" in ottale.

Un altro esempio: 11 111 111 è:

3 7 7

ovvero "377" in ottale.

Viceversa, l'ottale "211" rappresenta:

010 001 001

ovvero il binario "10001001".

L'ottale è stato tradizionalmente usato nei computer più vecchi i quali impiegavano vari numeri di bit che andavano da 8 fino a 64. Più recentemente, con il dominio dei microprocessori ad otto bit, il formato otto bit è diventato standard, ed è usata una rappresentazione ancora più pratica. Questa è chiamata *esadecimale*.

Nella rappresentazione esadecimale, un gruppo di quattro bit è codificato come un digit esadecimale. I digit esadecimali sono rappresentati dai simboli da 0 a 9 e dalle lettere A, B, C, D, E, F. Per esempio. "000" è rappresentato da "0", "0001" è rappresentato da "1" e "1111" è rappresentato dalla lettera "F" (vedere la Figura 1-8).

DECIMALE	BINARIO	ESADECIMALE	OTTALE
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Figura 1-8: Codice Esadecimale

Esempio: $\underbrace{1010}_{A}\ \underbrace{0001}_{1}$ in binario, è rappresentato da in esadecimale.

Esercizio 1-25: *Qual è la rappresentazione esadecimale di "10101010"?*

Esercizio 1-26: *Viceversa, qual è l'equivalente binario dell'esadecimale "FA"?*

Esercizio 1-27: *Qual è l'ottale di "01000001"?*

L'esadecimale offre il vantaggio di codificare otto bit in soli due digit.

Questo è più facile da visualizzare o memorizzare e più veloce da scrivere in un computer del suo equivalente binario. Perciò, nella maggior parte dei nuovi microcomputer, l'esadecimale è il metodo preferito di rappresentazione per gruppi di bit.

Naturalmente, ogni qualvolta l'informazione presente nella memoria ha un significato, come rappresentare un testo o numeri, l'esadecimale non è conveniente per rappresentare il significato di questa informazione quando è messo in evidenza per l'uso da parte dell'uomo.

Rappresentazione simbolica

La rappresentazione simbolica si riferisce alla rappresentazione esterna dell'informazione nella vera forma simbolica. Per esempio, i numeri decimali sono rappresentati come numeri decimali e non come sequenze di simboli esadecimali o bit. Allo stesso modo, il testo è rappresentato come tale.

Naturalmente, la rappresentazione simbolica è maggiormente pratica per chi la usa. È usata ogni qualvolta è disponibile un appropriato dispositivo di rappresentazione, come un display CRT (tubi a raggi catodici) oppure una stampante. (Un display CRT è uno schermo tipo televisione usato per rappresentare il testo o i grafici). Sfortunatamente, nei sistemi più piccoli come i microcomputer su scheda singola, è antieconomico fornire tali display, e l'utente è costretto alla comunicazione esadecimale con il computer.

Sommario delle Rappresentazioni Esterne

La rappresentazione simbolica delle informazioni è la più desiderabile poichè è la più naturale per un utente. Comunque, richiede un'interfaccia costosa costituita da una tastiera alfanumerica, più una stampante o un display CRT. Per questa ragione può non essere disponibile sui sistemi meno costosi. Allora viene impiegato un tipo alternativo di rappresentazione, e in questo caso l'esadecimale è la rappresentazione dominante. Soltanto in rari casi relativi ad un debugging (messa a punto) fine a livello di hardware o di software, è usata la rappresentazione binaria. Il *Binario* rappresenta direttamente il contenuto dei registri di memoria nel formato binario.
(L'utilità di un display binario diretto su un pannello frontale è sempre stato il soggetto di una accaldata controversia emozionale, che non sarà dibattuta qui).

Abbiamo visto come rappresentare le informazioni internamente ed esternamente. Ora esamineremo il microprocessore vero e proprio che manipolerà queste informazioni.

Esercizi addizionali

Esercizio 1-24: *Qual è il vantaggio del complemento a due sulle altre rappresentazioni usate per rappresentare i numeri con segno?*

Esercizio 1-25: *Come rappresentereste "1024" in binario diretto? In binario con segno? In complemento a due?*

Esercizio 1-26: *Cos'è il bit V? Il programmatore dovrebbe verificarlo dopo una addizione o una sottrazione?*

Esercizio 1-27: *Calcolate il complemento a due di "+ 16", "+ 17", "+ 18". 16", " 17", 18".*

Esercizio 1-28: *Mostrate la rappresentazione esadecimale del seguente testo, che è stato memorizzato internamente nel formato ASCII, senza parità: "MESSAGE".*

CAPITOLO 2

ORGANIZZAZIONE HARDWARE DELLO Z80

INTRODUZIONE

Per programmare ad un livello elementare, non è necessario capire in dettaglio la struttura interna del processore che si sta usando.

Comunque, per fare della programmazione efficiente, è richiesta una tale comprensione. Lo scopo di questo capitolo è di presentare i concetti hardware fondamentali necessari per capire il funzionamento del sistema Z80. Il sistema microcomputer completo non include solo l'unità microprocessore (qui lo Z80), ma anche altri componenti. Questo capitolo presenta lo Z80 propriamente detto, mentre gli altri dispositivi (principalmente di input output) saranno presentati in un capitolo separato (Capitolo 7).

Qui vedremo l'architettura fondamentale del sistema microcomputer e poi studieremo più da vicino l'organizzazione interna dello Z80.

In particolare, esamineremo i vari registri. Poi studieremo l'esecuzione del programma e il meccanismo di sequenza. Dal punto di vista dell'hardware, questo capitolo è soltanto una rappresentazione semplificata.

Lo Z80 fu progettato come sostituzione per l'Intel 8080, e per offrire capacità addizionali. In questo capitolo sarà fatto un certo numero di riferimenti al progetto 8080.

ARCHITETTURA DEL SISTEMA

L'architettura del sistema a microcomputer appare nella Figura 2-1.

L'unità microprocessore (MPU), che qui sarà lo Z80, appare alla sinistra dell'illustrazione. Esso implementa le funzioni di una *unità di elaborazione centrale* (CPU) su un chip: include una *unità aritmetico-logica* (ALU), più i suoi registri interni ed una *unità di controllo* (CU), col compito di sequenziare il sistema.

Il suo funzionamento sarà spiegato in questo capitolo.

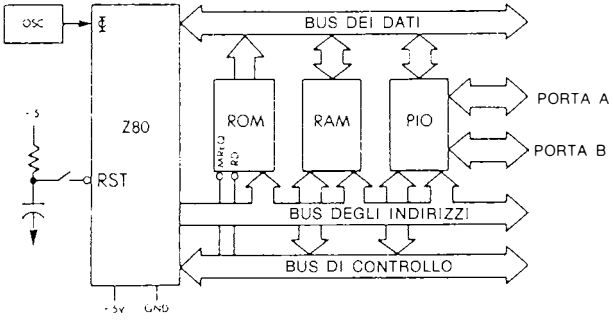


Figura 2-1: Sistema Z80 Standard

L'MPU fornisce tre *bus*: un *bus dei dati* bidirezionale ad 8 bit, che appare nella parte alta dell'illustrazione, un *bus degli indirizzi*, unidirezionale a 16 bit, ed un *bus di controllo* che appare nella parte bassa dell'illustrazione.

Descriviamo la funzione di ognuno di questi bus.

Il *bus dei dati* porta i dati che sono scambiati dai vari elementi del sistema. Tipicamente, porterà i dati dalla memoria all'MPU o dall'MPU alla memoria o dall'MPU ad un chip di input output. (Un chip di input/output è un componente col compito di comunicare con un dispositivo esterno).

Il *bus degli indirizzi* porta un indirizzo fornito dall'MPU, che sceglierà un registro interno entro uno dei chip annessi al sistema. Questo indirizzo specifica la sorgente, o la destinazione, dei dati che transiteranno lungo il bus dei dati.

Il *bus di controllo* porta i vari segnali di sincronizzazione richiesti dal sistema.

Dopo aver descritto lo scopo dei bus, ora colleghiamo i componenti addizionali richiesti per un sistema completo.

Ogni MPU richiede un riferimento di tempo preciso, che è fornito da un *clock* e da un *cristallo*. Nella maggior parte dei microprocessori più vecchi, l'oscillatore del clock è esterno all'MPU e richiede un chip extra. Nella maggior parte dei microprocessori recenti, l'oscillatore del clock è generalmente incorporato nell'MPU. Comunque, il cristallo al quarzo, a causa del suo volume, è sempre esterno al sistema. Il cristallo ed il clock appaiono alla sinistra del rettangolo MPU nella Figura 2-1.

Ora prestiamo la nostra attenzione agli altri elementi del sistema. Andando dalla sinistra alla destra dell'illustrazione, distinguiamo:

La ROM è la *memoria di sola lettura* e contiene il *programma* per il sistema. Il vantaggio della memoria ROM è che il suo contenuto è permanente e non si cancella ogni volta che il sistema è spento. Perciò, la ROM contiene sempre un programma *bootstrap* o *monitor* (le loro funzioni saranno spiegate più tardi) per permettere l'iniziale funzionamento del sistema. In una condizione di controllo di processo, quasi tutti i programmi risiederanno nella ROM, siccome non saranno probabilmente mai cambiati. In tale caso, colui che ne fa un uso industriale deve proteggere il sistema contro le mancanze di tensione; i programmi non devono essere in qualche modo cancellabili. Devono essere nella ROM.

Comunque, in applicazioni di tipo hobby, o in un ambiente di sviluppo del programma (quando il programmatore prova il suo programma) la maggior parte dei programmi risiederà nella RAM (memoria ad accesso casuale) in modo che possano essere facilmente cambiati.

Più tardi, possono rimanere nella RAM, o essere trasferiti nella ROM, se si vuole. Comunque, la RAM, è cancellabile. Il suo contenuto è perso quando viene tolta l'alimentazione.

La RAM è la memoria di lettura scrittura per il sistema. Nel caso di sistema di controllo, l'ammontare di RAM sarà tipicamente piccolo (solo per i dati). D'altra parte, in una condizione di sviluppo del programma, l'ammontare di RAM sarà grande, siccome conterrà i programmi più il software di sviluppo. Tutto il contenuto della RAM deve essere caricato prima dell'uso da un dispositivo esterno.

Alla fine, il sistema conterrà uno o più chip di interfaccia in modo che possa comunicare con il mondo esterno. Il chip di interfaccia più comunemente usato è il PIO o *parallel input/output* chip (chip d'ingresso/uscita parallelo). È quello mostrato nell'illustrazione. Questo PIO, come tutti gli altri chip nel sistema si collega a tutti e tre i bus e fornisce almeno due porte da 16 bit per comunicare con il mondo esterno. Per ulteriori dettagli su come veramente funzioni il PIO, riferitevi al libro C 201 oppure, per i particolari del sistema Z80, riferitevi al Capitolo 7 (Dispositivi input/output).

Tutti i chip sono collegati a tutti e tre i bus incluso il bus di controllo. Comunque, per chiarezza di illustrazione, i collegamenti tra il bus di controllo e questi vari chip non sono mostrati sullo schema.

I moduli funzionali che sono stati descritti non hanno necessariamente bisogno di risiedere su un chip LSI singolo. Infatti, noi potremmo usare *chip di combinazione*, che possono includere un ammontare limitato di ROM o RAM.

Tuttavia saranno richiesti altri componenti per costruire un sistema reale. In particolare i bus generalmente devono essere *con memoria di transito* (buffered). Può anche essere usata la *logica di decodifica* per i chip di memoria RAM e può darsi sia necessaria l'amplificazione di

oppurtuni *driver*. Questi circuiti ausiliari non saranno descritti qui in quanto non sono rilevanti nella programmazione. Il lettore interessato nelle tecniche di interfaccia e di assemblaggio specifiche si deve riferire al libro «Tecniche di Interfacciamento dei Microprocessori» C207.

DENTRO UN MICROPROCESSORE

La grande maggioranza dei chip microprocessori oggi sul mercato implementano la stessa architettura. Questa architettura «standard» sarà descritta qui. È mostrata nella Figura 2-2. I moduli di questo microprocessore standard saranno ora descritti in dettagli, da destra a sinistra.

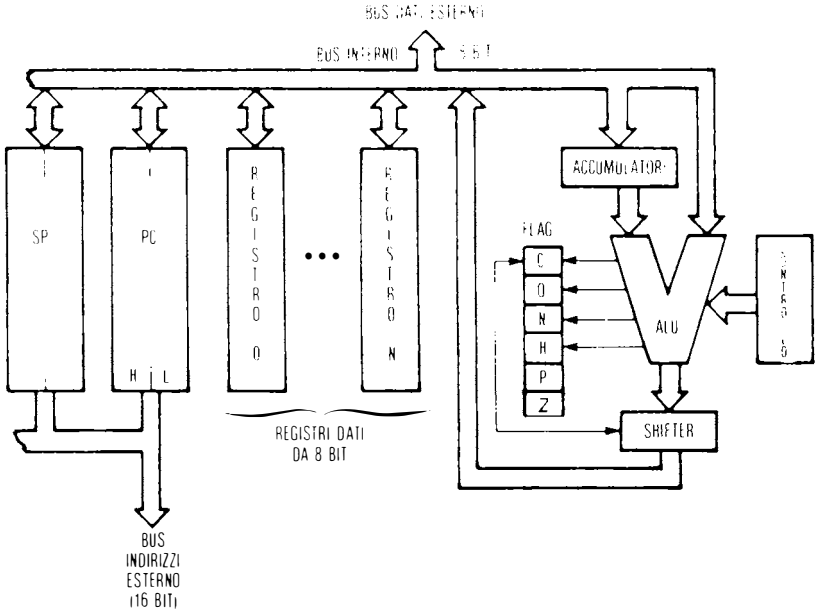


Figura 2-2: Architettura «Standard» del Microprocessore

Il *blocco di controllo* alla destra rappresenta l'unità di controllo che sincronizza l'intero sistema. Il suo ruolo sarà chiarito nella parte restante di questo capitolo.

L'ALU esegue operazioni logiche ed aritmetiche. Un registro speciale equipaggia uno degli input dell'ALU, l'input di sinistra qui. È chiamato accumulatore. (Possono esserci diversi accumulatori). L'accumulatore può essere riferito sia come input che come output (sorgente e destinazione) entro la stessa istruzione.

L'ALU deve anche fornire mezzi di *spostamento* e *di rotazione* (shift e rotate).

L'operazione di spostamento o scorrimento (shift) consiste nel muovere il contenuto di un byte di una o più posizioni a sinistra oppure a destra. Questo è illustrato nella Figura 2-3. Ogni bit è stato spostato a sinistra di una posizione. I dettagli degli spostamenti e delle rotazioni saranno presentati nel prossimo capitolo.

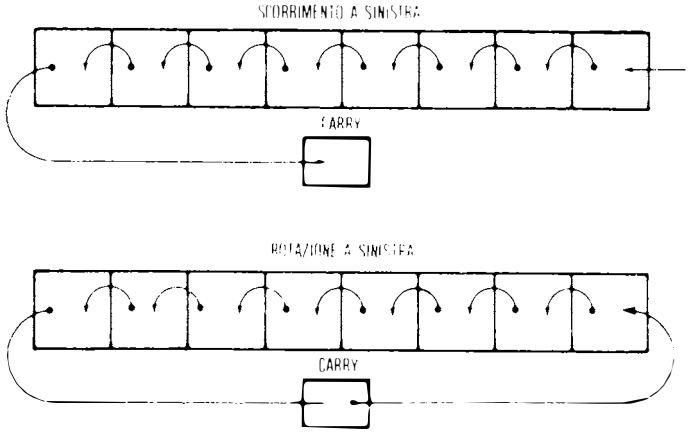


Figura 2-3: Spostamento e Rotazione

Il dispositivo a scorrimento (shifter) può essere all'output dell'ALU, come illustrato nella Figura 2-2, o può essere all'input dell'accumulatore.

Alla sinistra dell'ALU appaiono il registro di stato o dei flag.

Il loro ruolo è quello di immagazzinare condizioni eccezionali dentro il microprocessore. Il contenuto del registro dei flag può essere verificato tramite istruzioni specializzate, o può essere letto sul bus dei dati interni. Una istruzione *condizionale* causerà l'esecuzione di un nuovo programma, a seconda del valore di uno di questi bit.

Il ruolo dei bit di stato nello Z80 sarà esaminato più avanti in questo capitolo.

Posizionamento dei flag

La maggior parte delle istruzioni eseguite dal processore modificheranno alcuni o tutti i flag. È importante riferirsi sempre allo schema fornito dal produttore che elenca quali bit saranno modificati dalle istruzioni. Questo è essenziale per capire il modo in cui sta per essere eseguito un programma. Tale schema per lo Z80 è mostrato nell'Appendice.

I registri

Guardiamo ora alla Figura 2-2. Sulla sinistra dell'illustrazione, appaiono i registri del microprocessore. Si possono distinguere i *registri per scopi generali* (*general purpose*) ed i *registri degli indirizzi*.

I registri per scopi generali

I registri per scopi generali devono essere forniti in modo tale che l'ALU possa manipolare i dati ad alta velocità. A causa delle restrizioni sul numero di bit che è ragionevole fornire entro una istruzione, il numero dei registri (direttamente indirizzabili) è generalmente limitato a meno di otto. Ognuno di questi registri è una serie di otto flip-flop, collegati al bus dei dati interno bidirezionale.

Questi otto bit possono essere trasferiti simultaneamente al o dal bus dei dati. L'implementazione di questi registri con flip-flop MOS fornisce il più veloce livello di memoria disponibile, e si può accedere al loro contenuto entro decine di nanosecondi. I registri *interni* sono generalmente contrassegnati da 0 a n. Il ruolo di questi registri non è definito anticipatamente: di essi si dice che sono per "scopi generali". Possono contenere qualsiasi dato usato dal programma.

Questi registri per scopi generali saranno normalmente usati per immagazzinare i dati ad ot-

to bit. Su alcuni microprocessori, esistono i mezzi per manipolare *due* di questi registri alla volta. Sono allora chiamati "coppie di registri". Questa disposizione facilita l'immagazzinamento delle quantità a 16 bit, siano essi dati o indirizzi.

I registri degli indirizzi

I registri degli indirizzi sono registri a 16 bit intesi per l'immagazzinamento degli indirizzi. Spesso sono anche chiamati *contatori o puntatori dei dati*. Sono registri doppi, cioè, due registri ad otto bit. La loro caratteristica essenziale è quella di essere collegati al bus degli indirizzi. I registri degli indirizzi creano il bus degli indirizzi. Il bus degli indirizzi appare nella parte inferiore e alla sinistra dell'illustrazione nella Figura 2-4.

Il solo modo per caricare il contenuto di questi registri a 16 bit è tramite il bus dei dati. Saranno necessari due trasferimenti lungo il bus dei dati per trasferire 16 bit. Per differenziare tra la metà inferiore e la metà più alta di ogni registro, sono generalmente contrassegnati come L (low) o H (High), che denotano i bit da 0 a 7, e da 8 a 15 rispettivamente. Questo contrassegno è usato ogni qualvolta è necessario differenziare le metà di questi registri. La maggior parte dei microprocessori comprende almeno due registri degli indirizzi. "MUX" nella Figura 2-4 sta per multiplexer.

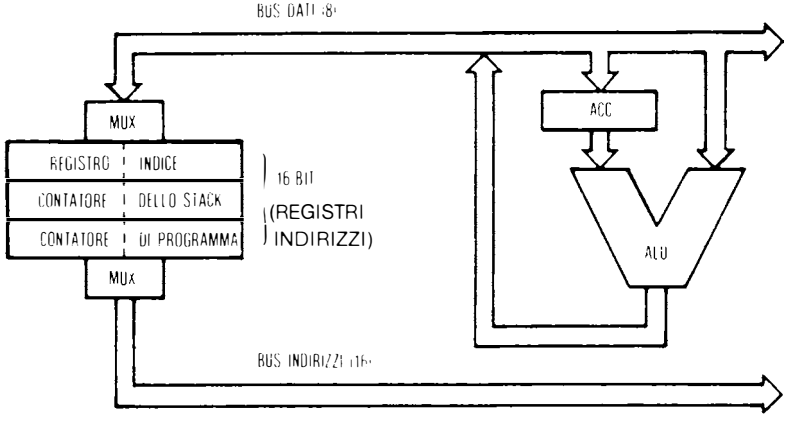


Figura 2-4: I registri degli indirizzi a 16 bit Formano il Bus degli Indirizzi

Il contatore di programma (PC)

Il *contatore di programma* deve essere presente in ogni processore. Contiene l'indirizzo dell'operazione successiva che deve essere eseguita. La presenza del contatore di programma è indispensabile e fondamentale all'esecuzione dei programmi. Il meccanismo dell'esecuzione dei programmi e la successione automatica compiuta con il contatore di programma saranno descritti al paragrafo successivo. Brevemente, l'esecuzione di un programma è normalmente sequenziale. Per accedere alla prossima istruzione, è necessario portarla dalla memoria nel microprocessore. Il contenuto del PC sarà depositato nel bus degli indirizzi, e trasmesso verso la memoria. La memoria leggerà poi il contenuto specificato da questo indirizzo e manderà indietro la parola corrispondente all'MPU. Questa è l'istruzione.

In alcuni microprocessori eccezionali, come l'F8 a due chip, non c'è un PC sul microprocessore. Questo non significa che il sistema non abbia un contatore di programma. Può accadere che il PC venga implementato direttamente sul chip della memoria, per ragioni di efficienza.

Puntatore dello Stack (SP)

Lo *stack* non è ancora stato introdotto e sarà descritto nel prossimo paragrafo. Nella maggior parte dei potenti microprocessori a scopo generale, lo stack è implementato via "software", cioè, entro la memoria. Un registro a 16 bit è dedicato al *Puntatore di stack* o SP, per tenere traccia della sommità di questo stack dentro la memoria. L'SP contiene l'indirizzo della sommità dello stack entro la memoria. Sarà mostrato che lo stack è indispensabile per gli interrupt (interruzioni) e per le subroutine (sottoprogrammi).

Registro Indice (IX)

L'indicizzazione è la capacità di indirizzare la memoria e non è sempre fornita nei microprocessori. Le varie tecniche di indirizzamento della memoria saranno descritte nel Capitolo 5. L'indicizzazione è la capacità di accedere a blocchi di dati nella memoria con una istruzione singola. Un *registro indice* conterrà tipicamente uno spostamento che sarà automaticamente sommato ad una base (oppure potrebbe contenere una base che sarà sommata ad uno spostamento). In breve, l'indicizzazione è usata per fare accedere a qualsiasi parola all'interno di un blocco di istruzioni.

Lo stack

Uno *stack* (catasta) è formalmente chiamata una struttura LIFO (last-in, first-out: l'ultimo che entra è il primo che esce). Uno stack è un gruppo di registri, o posizioni della memoria, assegnato a questa struttura di dati. La caratteristica essenziale di questa struttura è quella di essere una struttura *cronologica*. Il primo elemento introdotto nello stack è sempre alla parte più bassa dello stack. L'elemento depositato più recentemente nello stack è sulla sommità dello stack. L'analogia può essere fatta con una pila di piatti sul bancone di un ristorante. C'è un foro nel bancone con una molla nel fondo. I piatti sono impilati nel foro. Con questa organizzazione, è garantito che il piatto che è stato messo per primo nello stack (il più vecchio) è sempre in fondo. Quello che è stato posto più recentemente sullo stack è quello che è sulla cima. Questo esempio illustra anche un'altra caratteristica dello stack. Nell'uso normale, uno stack è accessibile solo tramite due istruzioni "push" e "pop" (o "pull").

L'operazione *push* consiste nel depositare un elemento sulla cima dello stack (due nel caso dello Z80).

L'operazione *pull* consiste nel rimuovere un elemento dello stack. Nel caso di un microprocessore, è l'*accumulatore* che sarà depositato sulla cima dello stack. Il *pop* risulterà in un trasferimento dell'elemento di cima dello stack nell'accumulatore. Possono esistere altre istruzioni speciali per trasferire la cima dello stack in altri registri specializzati, come il registro di stato. Sotto questo aspetto, lo Z80 è più versatile della maggior parte degli altri processori.

La disponibilità di uno stack è richiesta per rendere possibile tre possibilità di programmazione entro il sistema dei computer: sottoprogrammi (subroutine), interrupt e immagazzinamento temporaneo di dati. Il ruolo dello stack nelle subroutine programmi sarà spiegato nel Capitolo 3 (Tecniche Fondamentali di Programmazione). Il ruolo dello stack negli interrupt sarà spiegato nel Capitolo 6 (Tecniche di Input/output). Infine, il ruolo dello stack nel memorizzare i dati ad alta velocità sarà spiegato nel corso di programmi specifici di applicazione. A questo punto diremo soltanto che lo stack è un accessorio richiesto in ogni sistema computer. Uno stack può essere implementato in due modi.

1. Un numero fisso di registri può essere fornito entro lo stesso microprocessore. Questo è uno "stack hardware". Ha il vantaggio dell'alta velocità. Comunque, ha lo svantaggio di un numero limitato di registri.
2. La maggior parte dei microprocessori per scopi generali scelgono un altro approccio, lo stack software, per non restringere lo stack ad un numero molto piccolo di registri. Questo è l'approccio scelto per lo Z80. Nell'approccio software, un registro dedicato appositamente entro il microprocessore, qui il registro SP, immagazzina il puntatore di stack, cioè, l'indi-

rizzo dell'elemento in cima allo stack (o, a volte, l'indirizzo dell'elemento in cima allo stack più uno). Lo stack è poi implementato come un'area della memoria. Il puntatore dello stack richiederà perciò 16 bit per puntare in qualunque punto della memoria.

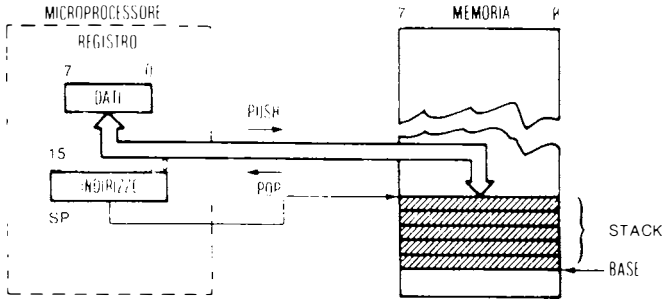


Figura 2-5: Le Due Istruzioni per la manipolazione dello Stack

Il ciclo di esecuzione delle Istruzioni

Ora, riferiamoci alla Figura 2-6. L'unità microprocessore appare alla sinistra, e la memoria appare alla destra. Il chip di memoria può essere una ROM oppure una RAM, o ogni altro chip che contiene memoria. La memoria è usata per immagazzinare istruzioni e dati. Qui, preleveremo una istruzione dalla memoria per illustrare il ruolo del contatore di programma. Noi presumiamo che il contatore di programma abbia contenuti validi. Ora contiene un indirizzo da 16 bit che è l'indirizzo della prossima istruzione da prelevare nella memoria. Ogni processore procede in tre cicli.

1 - prelevare la prossima istruzione (fetch)
2 – decodificare l'istruzione
3 – eseguire l'istruzione

Il fetch

Adesso seguiamo la sequenza. Nel primo ciclo, il contenuto del contatore di programmi è depositato sul bus degli indirizzi e portato alla memoria (sul bus degli indirizzi). Simultaneamente, un segnale di lettura può essere emesso sul bus di controllo del sistema, se è richiesto. La memoria riceverà l'indirizzo. Questo indirizzo è usato per specificare una locazione entro la memoria. Al ricevimento del segnale di lettura, le memoria decodificherà l'indirizzo che ha ricevuto, attraverso decodificatori interni, e sceglierà la locazione specificata dall'indirizzo. Alcune centinaia di nanosecondi più tardi, la memoria depositerà sul suo bus dei dati gli otto bit corrispondenti all'indirizzo specificato. Questa parola ad otto bit, è l'istruzione che vogliamo prelevare dalla memoria. Nella nostra illustrazione, questa istruzione sarà depositata alla cima del bus dei dati.

Riassumiamo brevemente la sequenza: il contenuto del contatore dei programmi è emesso sul bus degli indirizzi. È generato un segnale di lettura.

La *memoria cicla*, e forse 300 nanosecondi più tardi, l'istruzione all'indirizzo specificato è depositata sul bus dei dati (presupponendo una istruzione da un byte singolo). Il microprocessore allora legge il bus dei dati e deposita il suo contenuto in un registro interno specializzato, il registro IR. L'IR è il *registro delle istruzioni*: è largo 8 bit ed è usato per contenere l'istruzione appena prelevata dalla memoria. Il ciclo di prelievo è adesso completato. Gli 8 bit dell'istruzione sono ora fisicamente nello speciale registro interno dell'MPU, il registro IR. L'IR appare sulla sinistra della Figura 2-7. Non è accessibile al programmatore.

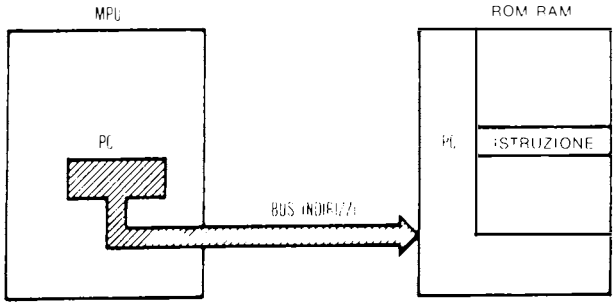


Figura 2-6: Prelievo di un'Istruzione dalla Memoria (Fetch)

Decodifica ed Esecuzione

Una volta che l'istruzione è contenuta nell'IR, l'unità di controllo del microprocessore decodificherà il contenuto e sarà capace di generare la sequenza corretta dei segnali interni e esterni per l'esecuzione dell'istruzione specificata. C'è, perciò, un breve ritardo di decodifica seguito da una fase di esecuzione, la cui lunghezza dipende dalla natura della istruzione specificata. Alcune istruzioni si eseguiranno interamente entro l'MPU. Altre istruzioni preleveranno o depositeranno dati da o nella memoria. Ecco perché le varie istruzioni dell'MPU richiedono varie lunghezze di tempo per l'esecuzione. Questa durata è espressa come un numero di cicli (di clock). Riferitevi all'Appendice per il numero di cicli richiesti da ogni istruzione.

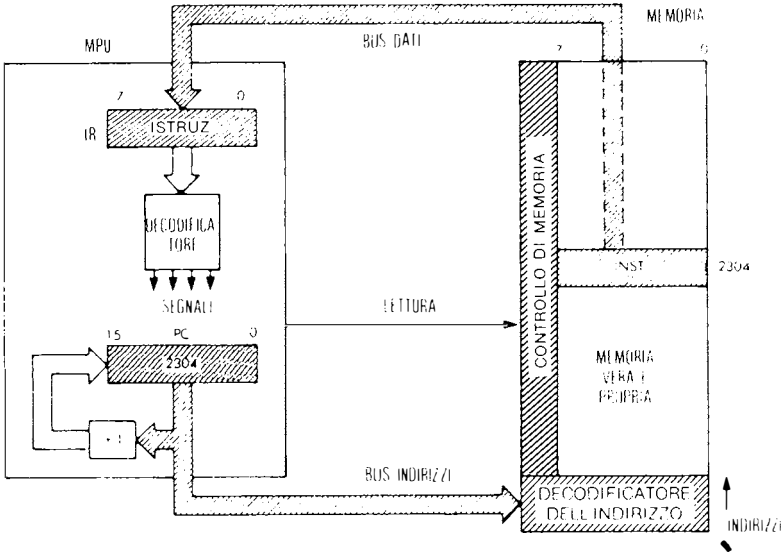


Figura 2-7: Sequenzializzazione automatica

Poiché possono essere usate varie velocità di clock, la velocità di esecuzione è normalmente espressa nel numero di cicli piuttosto che nel numero di nanosecondi.

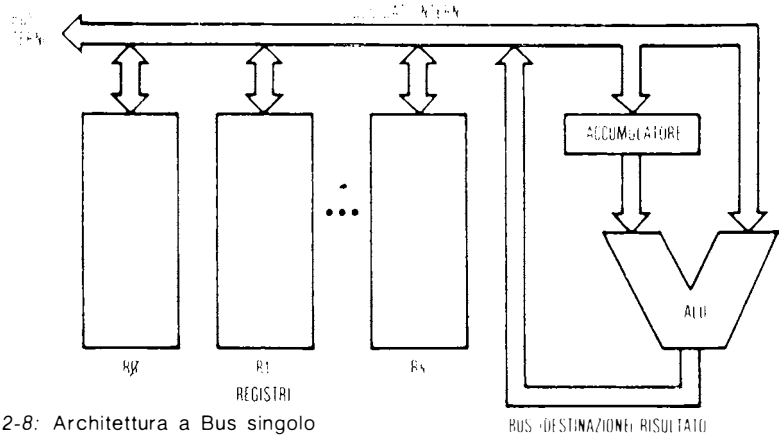


Figura 2-8: Architettura a Bus singolo

Prelievo dell'Istruzione successiva

Vi abbiamo descritto come una istruzione può essere prelevata dalla memoria usando un contatore di programma. Durante l'esecuzione di un programma, le istruzioni sono prelevate *in sequenza* dalla memoria. Deve essere perciò fornito di un meccanismo automatico per prelevare le istruzioni in sequenza. Questo compito è compiuto da un semplice incrementatore connesso al contatore di programma. Questo è illustrato nella Figura 2-7. Ogni volta che il contenuto del contatore di programma (nel fondo della illustrazione) è posto sul bus degli indirizzi, il suo contenuto sarà incrementato e riscritto nel contatore di programma. Come esempio, se il contatore di programma conteneva il valore "0", il valore "0" sarebbe uscito sul bus degli indirizzi. Allora il contenuto del contatore di programma sarebbe incrementato e il valore "1" sarebbe riscritto nel contatore di programma stesso. In questo modo, la prossima volta che viene usato il contatore di programma sarà prelevata l'istruzione all'indirizzo 1. Abbiamo così implementato un *meccanismo automatico per sequenziare le istruzioni.*

Deve essere sottolineato che le descrizioni dette sopra sono semplificate. In realtà, alcune istruzioni possono essere lunghe due o tre byte, cosicché saranno prelevati dalla memoria dei byte successivi. Comunque, il meccanismo è identico.

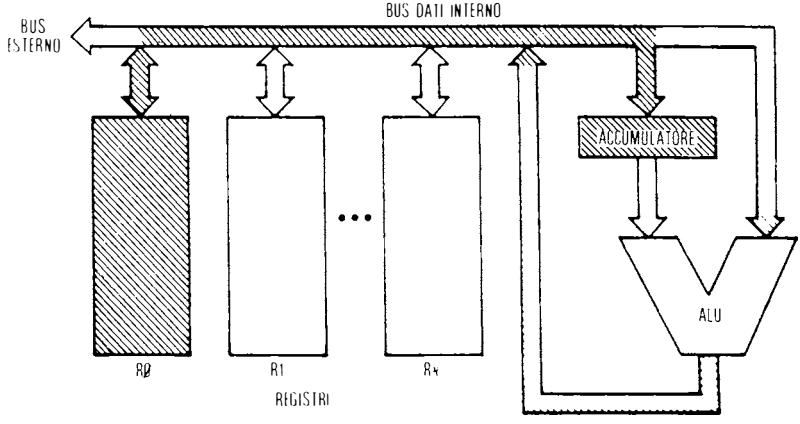


Figura 2-9: Esecuzione di una Addizione — R0 in ACC

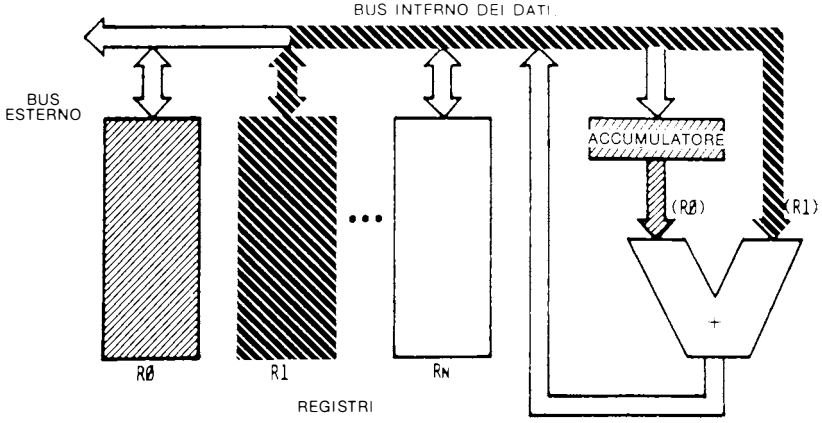


Figura 2-10: Addizione – Secondo Registro R1 in ALU

Il contatore di programma è usato per prelevare i byte successivi di una istruzione come per prelevare le istruzioni successive stesse. Il contatore di programma, insieme al suo incrementatore, fornisce un meccanismo automatico per puntare a locazioni di memoria successive.

Adesso eseguiremo una istruzione entro l'MPU (vedere la Figura 2-8).

Per esempio, una tipica istruzione sarà: R0=R0+R1. Questa significa: "somma (ADD) il contenuto di R0 e R1, ed immagazzina i risultati in R0".

Per eseguire questa operazione, il contenuto di R0 sarà letto dal registro R0, trasportato tramite il bus singolo all'input di sinistra dell'ALU, e immagazzinato là nel registro buffer (della memoria di transito). R1 sarà allora scelto e il suo contenuto sarà letto nel bus, poi trasferito all'input di destra dell'ALU. Questa sequenza è illustrata nelle Figure 2-9 e 2-10. A questo punto, l'input di destra dell'ALU è condizionato da R1, e l'input di sinistra è condizionato dal registro della memoria di transito, che contiene il precedente valore di R0. L'operazione può essere eseguita. L'addizione è eseguita dall'ALU ed i risultati appaiono sull'output dell'ALU, nell'an-

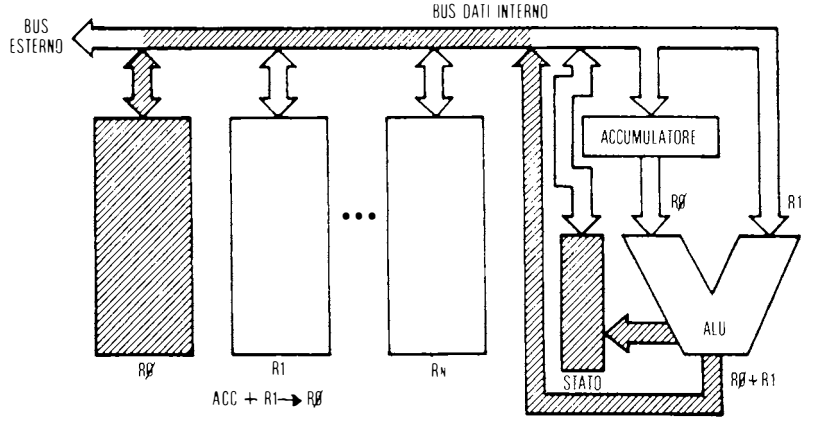


Figura 2-11: Il risultato è generato e va in R0

golo più basso a destra della Figura 2-11. I risultati saranno depositati sul bus singolo, e saranno trasferiti indietro all'R0.

In pratica, questo significa che il latch dell'input di R0 sarà abilitato, in modo che si possa scrivere in esso. L'esecuzione della istruzione è ora completa. I risultati dell'addizione sono nell'R0. Si dovrebbe osservare che il contenuto di R1 non è stato modificato da questa operazione. Questo è un principio generale: il contenuto di un registro, o qualsiasi locazione della memoria di lettura scrittura, non è modificato da una operazione di lettura.

Il registro buffer sull'input di sinistra dell'ALU era necessario per *memorizzare* il contenuto dell'R0, in modo che un singolo bus potesse essere usato nuovamente per un altro trasferimento. Comunque, rimane un problema.

Il problema della corsa critica

L'organizzazione semplice mostrata nella Figura 2-8 non funzionerà correttamente.

Domanda: *Cos'è il problema del timing (Temporizzazione)?*

Risposta: Il problema è costituito dal fatto che il risultato emesso dall'ALU sarà ridepositato sul bus singolo. Non si propagherà soltanto nella direzione dell'R0 ma lungo tutto il bus. In particolare, ripristinerà l'input di destra dell'ALU, cambiando il risultato che ne esce alcuni nanosecondi più tardi. Questa è una *corsa critica*. L'output dell'ALU deve essere isolato dal suo input (vedere la Figura 2-12).

Sono possibili diverse soluzioni, le quali isoleranno l'input dell'ALU dall'output. Deve essere usato un registro buffer. Il registro buffer potrebbe essere posto sull'output dell'ALU, o sul suo input. È generalmente posto sull'input dell'ALU. Qui sarebbe posto sul suo input destro. La memorizzazione di transito del sistema è ora sufficiente per una operazione corretta. Sarà mostrato più tardi in questo capitolo che se il registro di sinistra che appare in questa illustrazione deve essere usato come un accumulatore (permettendo l'uso di istruzioni di un byte di lunghezza), allora anche l'accumulatore richiederà un buffer (memoria di transito), come mostrato nella Figura 2-13.

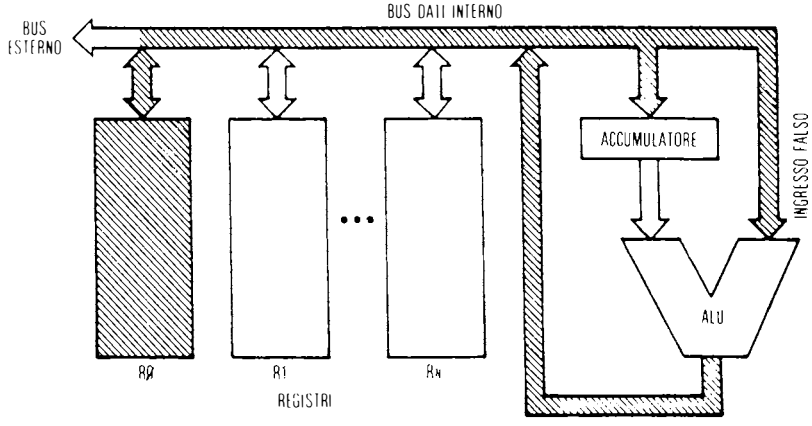


Figura 2-12: Il Problema della Corsa Critica

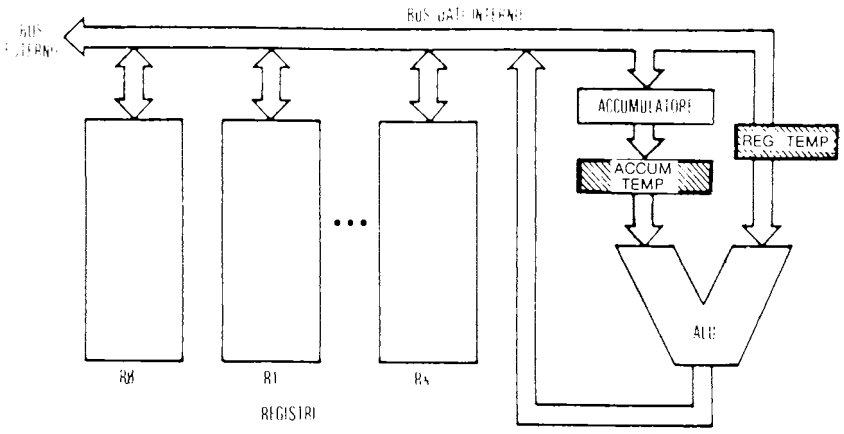


Figura 2-13: Sono Richiesti due Buffer

ORGANIZZAZIONE INTERNA DELLO Z80

Sono stati definiti i termini necessari per capire gli elementi interni del microprocessore. Adesso esamineremo più dettagliatamente lo Z80, e descriveremo le sue capacità. L'organizzazione interna dello Z80 è mostrata nella Figura 2-14. Questo schema presenta una descrizione logica del dispositivo. Possono esistere delle interconnessioni addizionali ma non sono mostrate. Esaminiamo lo schema da destra a sinistra. Nella parte destra dell'illustrazione, l'*unità aritmetico logica* (l'ALU) può essere riconosciuta dalla sua caratteristica forma a "V". Il registro dell'accumulatore, che è stato descritto nel paragrafo precedente, è identificato come A sul percorso dell'input destro dell'ALU. È stato mostrato nella sezione precedente che l'accumulatore dovrebbe essere fornito di un *registro buffer*. Questo è il registro definito ACT (accumulatore temporaneo). Qui, anche l'input sinistro è fornito di un *registro temporaneo*, chiamato TMP.

Il funzionamento dell'ALU diventerà chiaro al prossimo paragrafo, dove descriveremo l'esecuzione delle istruzioni vere.

Il *registro dei flag* è chiamato "F" nello Z80, ed è mostrato alla destra del registro accumulatore. Il contenuto del registro dei flag è essenzialmente condizionato dall'ALU, ma sarà mostrato che anche alcuni dei suoi bit possono essere condizionati da altri moduli o eventi.

L'accumulatore ed il registro dei flag sono mostrati come registri doppi contrassegnati rispettivamente A, A' e F, F'.

Questo è perché lo Z80 è internamente rifornito di due serie di registri: A + F, e A' + F'.

Comunque, solo *una* serie di questi registri può essere usata in qualsiasi momento. È fornita una istruzione speciale per scambiare il contenuto di A e F con A' e F'. Per semplificare le spiegazioni, nella maggior parte degli schemi che seguono, saranno mostrati solo A e F. Il lettore dovrebbe ricordare che ha la possibilità di commutare alla serie A' e F' del registro alternativo se lo si desidera.

Il ruolo di ogni flag nel registro di flag sarà descritto nel Capitolo 3 (Tecniche Fondamentali di Programmazione).

Al centro dell'illustrazione è mostrato un grande blocco di registri. In cima al blocco dei registri, possono essere riconosciuti due gruppi identici. Ogni gruppo include sei registri contrassegnati B, C, D, E, H, L. Questi sono i *registri ad otto bit per scopi generali* dello Z80. Ci sono due peculiarità dello Z80, rispetto al microprocessore standard, che sono state descritte all'inizio di questo capitolo.

Primo, lo Z80 è fornito di *due* bank di registri, cioé, due gruppi identici di 6 registri. Possono

essere usati solo sei registri alla volta. Comunque, sono fornite istruzioni speciali per commutare tra i due bank di registri.

Perciò, un bank si comporta come una memoria interna, mentre l'altro si comporta come una specie operante di registri interni. Nel prossimo capitolo saranno descritti i possibili usi di questo accessorio speciale.

Concettualmente, si presume d'ora in poi che ci sono solo sei registri che funzionano, B, C, D, E, H e L e il secondo bank di registri sarà temporaneamente trascurato per evitare confusione.

Il simbolo MUX che appare sopra il bank della memoria è l'abbreviazione di *multiplexer*. I dati che provengono dal bus dei dati interno saranno portati attraverso i multiplexer al registro scelto. Comunque, solo uno di questi registri alla volta può essere collegato al bus dei dati interni. Una seconda caratteristica di questi sei registri, oltre ad essere dei registri ad otto bit per scopi generali, è quella di essere forniti di un collegamento al *bus degli indirizzi*. Ecco perché sono stati raggruppati in *coppie*. Per esempio, il contenuto di B e C può essere portato simultaneamente nel bus degli indirizzi a 16 bit che appare al fondo dell'illustrazione. Come risultato, questo gruppo di 6 registri può essere usato per immagazzinare i dati ad otto bit oppure i *puntatori* a 16 bit che possono servire per l'indirizzamento della memoria.

Il terzo gruppo di registri, che appare sotto i due precedenti nel mezzo della Figura 2-14 contiene quattro registri degli indirizzi "puri". Come in ogni microprocessore, troviamo il PC e l'SP. Ricordatevi che il program counter (PC) contiene l'indirizzo della prossima istruzione da eseguire.

Il puntatore dello stack (stack pointer) punta alla cima dello stack nella memoria. Nel caso dello Z80, lo stack pointer punta all'*ultima vera entrata* nello stack. (In altri microprocessori, il puntatore dello stack punta proprio sopra l'ultima entrata). Poi, lo stack cresce *"verso il basso"*, cioè, verso gli indirizzi più bassi.

Questo significa che il puntatore dello stack deve essere *decrementato* ogni volta che una nuova parola è *spinta* sullo stack. Viceversa, ogni qual volta una parola è *rimossa* dallo stack, il puntatore dello stack deve essere *incrementato* di una parola. Nel caso dello Z80 il "push" e il "pop" intendono sempre *due* parole allo stesso tempo, in modo che il contenuto del puntatore dello stack sarà decrementato o incrementato di due.

Guardando ai rimanenti due registri di questo gruppo di quattro registri, troviamo un nuovo tipo di registro che non è stato ancora descritto: due *registri indice*, contrassegnati IX (Registro Indice X) e IY (Registro Indice Y). Questi due registri sono forniti di un sommatore speciale mostrato come un ALU a forma di V in miniatura alla destra di questi registri nella Figura 2-14. Un byte portato lungo il bus dei dati interni, può essere aggiunto al contenuto di IX o di IY. Questo byte è chiamato lo *spostamento* quando si usa una istruzione indicizzata. Sono fornite istruzioni speciali, le quali sommeranno automaticamente questo spostamento al contenuto di IX o IY e genereranno un indirizzo. Questo è chiamato *indicizzazione*. Permette un comodo accesso ad ogni blocco sequenziale di dati. Questa caratteristica importante sarà descritta nel Capitolo 5 sulle tecniche di indirizzo. Infine, un blocco speciale contrassegnato "$\pm$ 1" appare sotto a sinistra del blocco dei registri. Questo è un incremento/decremento. Il contenuto di ognuno di questi quattro registri che appartengono al gruppo che abbiamo appena descritto (i registri degli "indirizzi puri") può essere automaticamente incrementato o decrementato ogni volta che essi depositano un indirizzo nel bus degli indirizzi interni. Questo è un accessorio essenziale per rendere effettivi alcuni loop (cicli) di programma automatizzati, che saranno descritti nel prossimo paragrafo. Usando questa caratteristica, sarà possibile accedere a successive posizioni della memoria in modo conveniente.

Adesso, spostiamoci alla sinistra dell'illustrazione. È mostrata una coppia di registri isolata alla sinistra. Il registro I è chiamato *il registro di indirizzo della pagina di interrupt*. Il suo ruolo sarà descritto nel paragrafo sugli interrupt del Capitolo 6 (Tecniche d'input/output). È usato soltanto in un modo speciale dove è generata una chiamata indiretta alla posizione della memoria in risposta ad un interrupt.

Il registro I è usato per immagazzinare la parte di alto ordine dell'indirizzo indiretto. La parte più bassa dell'indirizzo è fornita dal dispositivo che ha generato l'interrupt.

Il registro R è il *registro di refresh della memoria*. È fornito per il refresh ("rinfresco") automatico delle memorie dinamiche. Tradizionalmente, questo registro è stato posto all'esterno

del microprocessore, poiché è associato con la memoria dinamica. È una comoda caratteristica che minimizza l'ammontare di hardware esterno per alcuni tipi di memorie dinamiche. Non sarà usato qui per nessuno scopo di programmazione, siccome è essenzialmente una caratteristica hardware. Comunque, è per esempio, possibile usarlo come un software clock.

Adesso spostiamoci alla parte più a sinistra dell'illustrazione. In questo punto è posta la sezione di controllo del microprocessore. Dall'alto al basso, troviamo innanzi tutto il *registro di istruzione* IR, il quale conterrà l'istruzione che deve essere eseguita. Il registro IR è totalmente diverso dalla coppia di registri "I, R" descritti sopra. L'istruzione è ricevuta dalla memoria per mezzo del bus dei dati, è trasmessa lungo il bus dei dati interno ed alla fine è depositata nel registro di istruzione. Sotto il registro di istruzione appare il *decodificatore* che invierà segnali al sequenziatore-controllore e causerà l'esecuzione dell'istruzione dentro e fuori il microprocessore.

La *sezione di controllo* genera e gestisce il bus dei controlli il quale appare nella parte bassa della illustrazione.

I tre bus amministrati e generati dal sistema, cioè, il bus dei dati, il bus degli indirizzi e il bus dei controlli, si propagano fuori del microprocessore attraverso i suoi pin.

I collegamenti esterni sono mostrati nella parte più a destra dell'illustrazione. I bus sono isolati dall'esterno attraverso i buffer (le memorie di transito) mostrati nella Figura 2-14.

Sono stati così descritti tutti gli elementi logici dello Z80.

Per iniziare a scrivere programmi, non è necessario capire il funzionamento dettagliato dello Z80.

Comunque, per il programmatore che desidera scrivere codici efficienti, la velocità del programma e la sua dimensione dipenderanno dalla scelta corretta dei registri come dalla scelta corretta delle tecniche. Per fare una scelta corretta, è necessario capire come sono eseguite le istruzioni dentro il microprocessore. Perciò, qui esamineremo l'esecuzione di istruzioni tipiche all'interno dello Z80 per dimostrare il ruolo e l'uso dei registri interni e dei bus.

FORMATI DELLE ISTRUZIONI

Le istruzioni dello Z80 sono elencate nell'Appendice. Le istruzioni possono essere disposte sistematicamente in uno, due, tre o quattro byte. Una istruzione specifica l'operazione che deve essere compiuta dal microprocessore. Da un punto di vista semplificato, ogni istruzione deve essere rappresentata come un *codice operativo* (opcode) seguito da un opzionale letterale o campo di indirizzo che comprende una o due parole. Il campo di codice operativo specifica l'operazione da realizzare. Nella più stretta terminologia dei computer, il codice operativo rappresenta solo quei bit che specificano l'operazione da compiere, esclusivi dei puntatori dei registri che potrebbero essere necessari. Nel mondo dei microprocessori è conveniente chiamare codice operativo lo stesso codice di operazione, così come qualsiasi puntatore dei registri che possa incorporare. Per efficienza, questo codice operativo generalizzato deve risiedere in una parola di otto bit (questo è il fattore limitante sul numero di istruzioni disponibili in un microprocessore).

L'8080 usa istruzioni che possono essere lunghe uno, due o tre byte (vedere la Figura 2-15). Comunque, lo Z80 è fornito di istruzioni indicizzate addizionali, che richiedono un byte supplementare. Nel caso dello Z80, in generale i codici operativi sono lunghi un byte eccetto che per istruzioni speciali che richiedono un codice operativo di due byte.

Alcune istruzioni richiedono che un byte di dati segua il codice operativo. In tale caso, l'istruzione sarà una istruzione di due byte, il cui secondo byte è fatto di dati (eccetto che per l'indicizzazione, la quale aggiunge un byte extra).

In altri casi, l'istruzione potrebbe richiedere la specificazione di un indirizzo. Un indirizzo richiede 16 bit e perciò, due byte. In quel caso, l'istruzione sarà una istruzione di tre byte o di quattro byte.

Per ogni byte di una istruzione, l'unità di controllo dovrà compiere un prelievo da memoria, che richiederà quattro cicli di clock. Più corta è l'istruzione, più veloce è l'esecuzione.

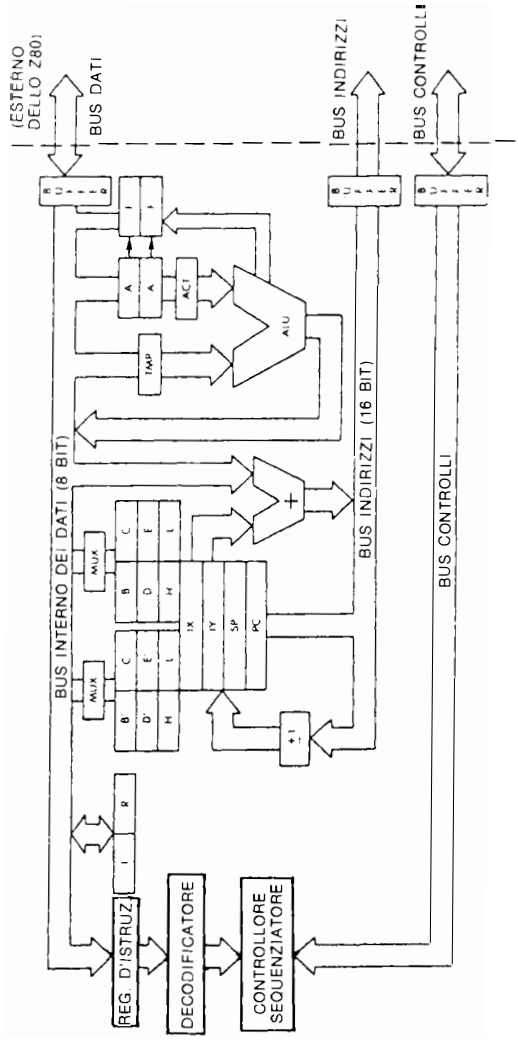


Figura 2-14: Organizzazione interna dello Z80

Istruzioni di una sola parola

Le istruzioni di una parola sono le più veloci e sono preferite dai programmatori. Una tale istruzione tipica per lo Z80 è:

L D r, r'

Questa istruzione significa "trasferisci il contenuto del registro r' in r." Questa è una tipica operazione "registro-registro". Ogni microprocessore deve essere fornito di tali istruzioni, che permettono al programmatore di trasferire le informazioni da ognuno dei registri della macchina in un altro. Le istruzioni che si riferiscono a registri speciali della macchina, come l'accumulatore o altri registri a scopi speciali, possono avere un codice operativo speciale.

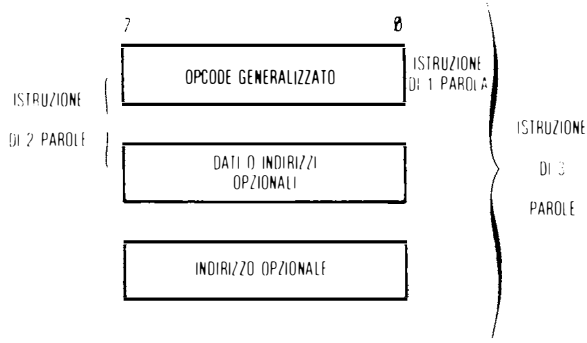


Figura 2-15: Formati Tipici delle Istruzioni

Dopo l'esecuzione dell'istruzione precedente, il contenuto di r sarà uguale al contenuto di r'. Il contenuto di r' non dovrebbe essere stato modificato dalla operazione di lettura. Ogni istruzione deve essere rappresentata internamente in un formato binario.

La rappresentazione "L D r, r'" è simbolica o *mnemonica*. È chiamata la rappresentazione in *linguaggio assembly* di una istruzione. È semplicemente intesa come una conveniente rappresentazione simbolica della vera codifica binaria per quell'istruzione. Il codice binario che rappresenterà questa istruzione all'interno della memoria è: 01 D D D S S S (bit da 0 a 7).

Questa rappresentazione è tuttavia parzialmente simbolica. Ognuna delle lettere S e D sta per un bit binario. I tre D, "DDD", rappresentano i tre bit che puntano al registro di *destinazione*. Tre bit permettono la scelta di uno degli otto registri possibili. I codici per questi registri appaiono nella Figura 2-16. Per esempio, il codice per il registro B è "000", il codice per il registro C è "001", e così via.

Analogamente, "SSS" rappresenta i tre bit che puntano al registro di *sorgente*. La convenzione qui è che il registro r' è la sorgente, e che il registro r è la destinazione. Il collocamento dei bit nella rappresentazione binaria di una istruzione non è inteso per la convenienza del programmatore, ma per la convenienza della sezione di controllo del microprocessore, la quale deve decodificare ed eseguire l'istruzione. Comunque, la rappresentazione in *linguaggio assembly* è intesa per la comodità del programmatore. Potrebbe essere provato che L D r, r' dovrebbe in realtà significare: "Trasferisci il contenuto di r in r'".

Comunque, in questo caso, la convenzione è stata scelta per mantenere la compatibilità con la rappresentazione binaria. Naturalmente essa è arbitraria.

Esercizio 2-1: *Scrivete di seguito il codice binario che trasferirà il contenuto del registro C nel registro B. Consultate la Figura 2-16 per i codici che corrispondono a C e B.*

Un altro semplice esempio di una istruzione ad una parola è

ADD A, r

Questa istruzione sommerà il contenuto di un registro specificato (r) all'accumulatore (A). Simbolicamente, questa operazione può essere rappresentata da: A = A + r. Nell'Appendice C può essere verificato che la rappresentazione binaria di questa istruzione è:

1 0 0 0 0 S S S

dove SSS specifica il registro da addizionare all'accumulatore. Di nuovo, i codici dei registri appaiono nella Figura 2-16.

Esercizio 2-2: *Qual è il codice binario dell'istruzione che addizionerà il contenuto del registro D all'accumulatore?*

CODICE	REGISTRO
0 0 0	B
0 0 1	C
0 1 0	D
0 1 1	E
1 0 0	H
1 0 1	L
1 1 0	- (MEMORIA)
1 1 1	A

Figura 2-16: I codici dei registri

Un'istruzione di due parole

ADD A, n

Questa semplice istruzione di due parole addizionerà il contenuto del secondo byte dell'istruzione all'accumulatore.

Il contenuto della seconda parola dell'istruzione è un letterale. Si tratta di dati che vengono trattati come otto bit senza nessun significato particolare. Potrebbero essere un carattere o dati numerici. Questo è irrilevante all'operazione. Il codice per questo ordine è:

1 1 0 0 0 1 1 0 seguito dal byte "n" di otto bit

Questa è un'operazione *immediata*. Nella maggior parte dei linguaggi della programmazione, "immediata" significa che la parola seguente, o le parole, entro l'istruzione contiene dei dati che non dovrebbero essere *interpretati* (come invece lo è il codice operativo). Significa che le una o due parole successive devono essere trattate come un campo di indirizzo.

L'unità di controllo è programmata per "sapere" quante parole ha ogni istruzione. Perciò, preleverà ed eseguirà sempre il numero esatto di parole per ogni istruzione. Comunque, più lungo è il possibile numero di parole per l'istruzione e più complessa è la decodifica per l'unità di controllo.

Un'istruzione di tre parole

LD A, (nn)

L'istruzione richiede tre parole. Significa: "Carica l'accumulatore dall'indirizzo della memoria specificato nei due byte successivi dell'istruzione". Poiché gli indirizzi sono lunghi 16 bit, richiedono due parole. Nel binario, questa istruzione è rappresentata da:

0 0 1 1 1 0 1 0:	8 bit per l'opcode
Indirizzo basso:	8 bit per la parte più bassa dell'indirizzo
Indirizzo alto:	8 bit per la parte più alta dell'indirizzo

ESECUZIONE DELLE ISTRUZIONI ENTRO LO Z80

Abbiamo visto che tutte le istruzioni sono eseguite in tre fasi: PRELIEVO, DECODIFICA, ESECUZIONE. Adesso abbiamo bisogno di introdurre alcune definizioni. Ognuna di queste fasi richiederà diversi cicli di clock. Lo Z80 esegue ogni fase in uno o più cicli logici, chiamati "cicli di macchina". Il ciclo di macchina più corto dura tre cicli di clock.

L'accesso alla memoria richiede quattro cicli di clock. Poiché ogni istruzione deve essere prima prelevata dalla memoria, l'istruzione più veloce richiederà quattro cicli di clock. La maggior parte delle istruzioni ne richiederà di più.

Ogni ciclo di macchina è contrassegnato come M1, M2, ecc., e richiederà tre o più cicli di clock, o "stati", contrassegnati T1, T2, ecc.

LA FASE DI PRELIEVO (FETCH)

La fase di prelievo di una istruzione è eseguita durante i primi tre stati del ciclo di macchina M1; sono chiamati T1, T2 e T3. Questi tre stati sono comuni a tutte le istruzioni del microprocessore, poiché tutte le istruzioni devono essere prelevate prima dell'esecuzione. Il meccanismo di FETCH (PRELIEVO) è il seguente:

T1 : PC OUT

La prima fase è quella di presentare l'indirizzo dell'istruzione che segue alla memoria.

Questo indirizzo è contenuto nel contatore di programma (PC). Come primo passo di ogni ricerca (fetch) di istruzione si mette il contenuto del PC sul bus degli indirizzi (vedere la Figura 2-17). A questo punto è presentato un indirizzo alla memoria, e il codificatore degli indirizzi di memoria, decodificherà questo indirizzo per scegliere la giusta locazione dentro la memoria. Trascorreranno diverse centinaia di ns (un nanosecondo è 10^{-9} secondi) prima che il contenuto della posizione di memoria selezionata diventi disponibile sui pin (piedini) di uscita della memoria, che sono collegati al bus dei dati.

È peculiare del progetto standard di computer sfruttare il tempo di lettura della memoria per eseguire un'operazione all'interno del microprocessore. Questa operazione consiste nell'incrementare il contatore di programma:

T2 : PC = PC + 1

Mentre è in attc il ciclo di lettura in memoria, il contenuto del PC è aumentato di uno (vedere la Figura 2-18). Alla fine dello stato T2, il contenuto della memoria è disponibile e può essere trasferito all'interno del microprocessore.

T3 : INST dentro IR

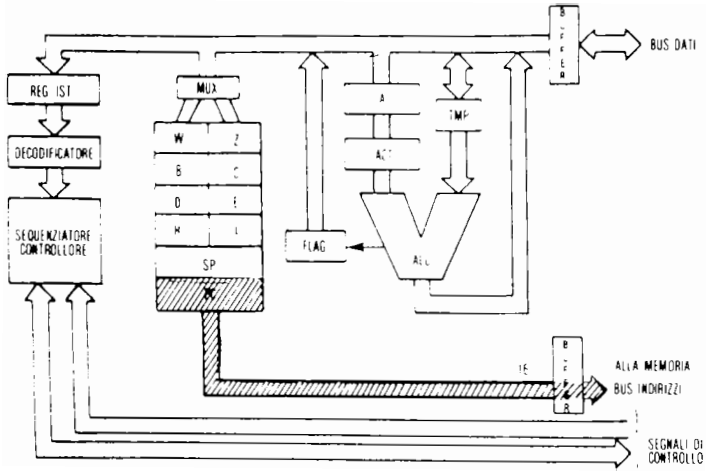


Figura 2-17: Prelievo dell'istruzione. Il contenuto di PC è inviato in memoria

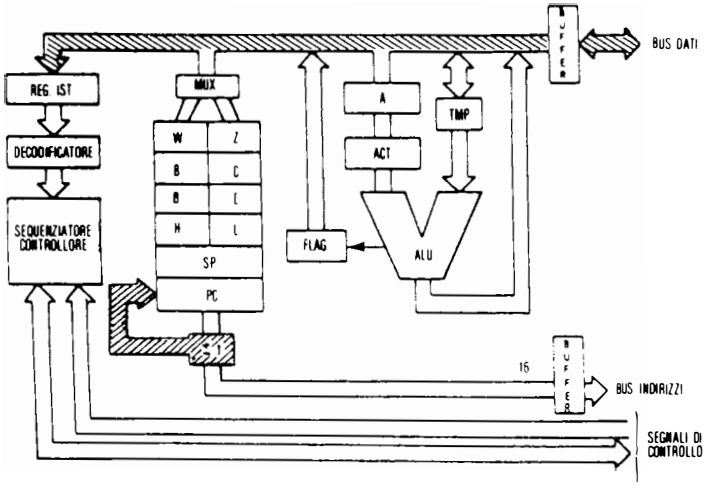


Figura 2-18: PC è incrementato

LE FASI DI DECODIFICA E DI ESECUZIONE

Durante lo stato T3, l'istruzione che è stata letta dalla memoria, è depositata sul bus dei dati e trasferita nel registro delle istruzioni dello Z80, dove sarà decodificata.

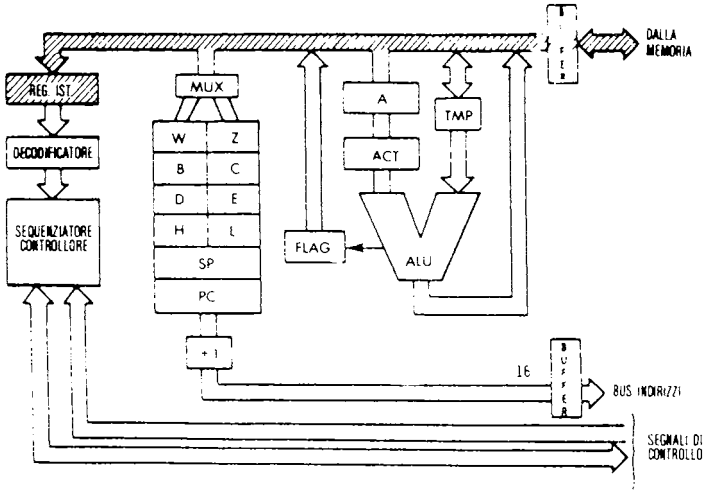


Figura 2-19: L'istruzione va dalla memoria al registro IR

Dovrebbe essere notato che sarà sempre richiesto lo stato T4 di M1. Una volta che l'istruzione è stata depositata nell'IR durante il T3, è necessario *decodificarla* ed *eseguirla*. Ciò richiederà almeno un ulteriore stato del ciclo di macchina, T4.

Un numero limitato di istruzioni richiede uno stato extra di M1 (stato T5). Tale stato sarà saltato dal processore per la maggior parte delle istruzioni. Ogni volta che l'esecuzione di una istruzione richiede più di M1, cioè M1, M2 o più cicli, la transizione sarà direttamente dallo stato T4 di M1 nello stato T1 di M2. Esaminiamo un esempio. La dettagliata sequenza interna per ogni esempio è mostrata nelle tabelle della Figura 2-27. Poichè queste tabelle non sono state fornite per lo Z80, sono invece usate le tabelle dell'8080. Forniscono una comprensione approfondita dell'esecuzione dell'istruzione.

LD D, C

Questo corrisponde a MOV r1, r2 per l'8080. Riferitevi alla linea 1 della Figura 2-27. Per coincidenza, risulta che in questo esempio il registro destinazione sia chiamato "D".

Il trasferimento è illustrato nella Figura 2-20.

Questa istruzione è stata descritta al paragrafo precedente. Trasferisce il contenuto del registro C, indicato con "C", nel registro D.

I primi tre stati del ciclo M1 sono usati per prelevare l'istruzione dalla memoria. Alla fine di T3, l'istruzione è nell'IR, il Registro di Istruzione, dove può essere decodificata (vedere la Figura 2-19).

Durante T4: (SSS) → TMP

Il contenuto di C è depositato in TMP (vedere la Figura 2-21).

Durante T5: (TMP) → DDD

Il contenuto di TMP è depositato in D. Questo è mostrato nella Figura 2-22.

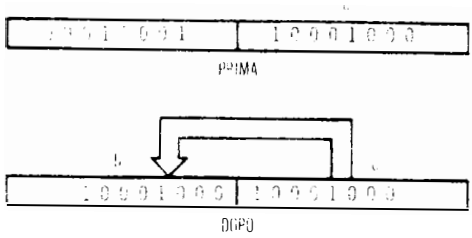


Figura 2-20: Trasferimento di C in D

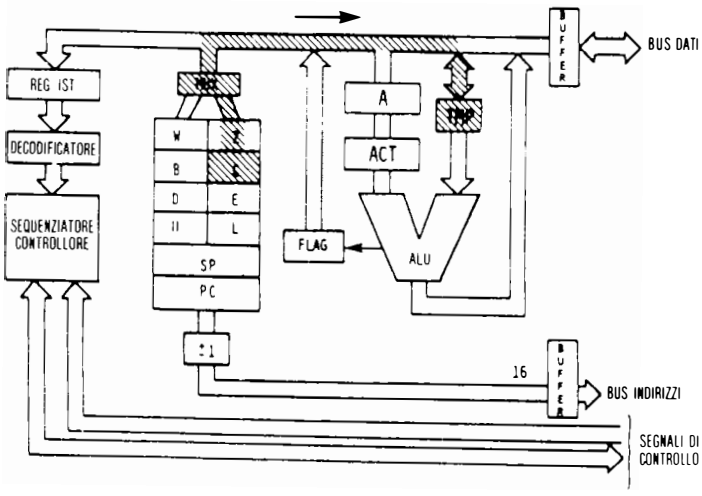


Figura 2-21: Il contenuto di C è depositato in TMP

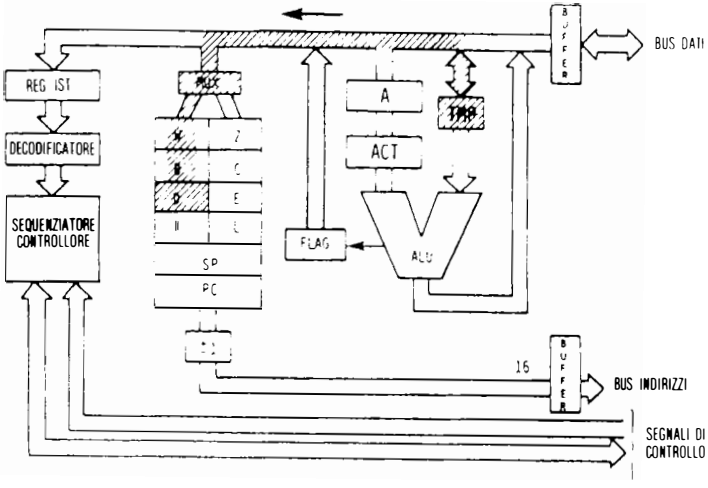


Figura 2-22: Il contenuto di TMP è depositato in D

L'esecuzione dell'istruzione è ora completa. Il contenuto del registro C è stato trasferito nel registro specificato destinazione D. Questo completa l'esecuzione dell'istruzione. Gli altri cicli di macchina M2, M3, M4 e M5 non saranno necessari e l'esecuzione si ferma con M1.

È possibile calcolare la durata di questa istruzione in modo facile. La durata di ogni stato per lo Z80 standard, è la durata del clock: 500 ns. La durata di questa istruzione è la durata di 5 stati, ovvero $5 \times 500 = 2500\,ns = 2.5\,\mu s$.

Domanda: *Perchè questa istruzione richiede due stati, T4 e T5, per trasferire il contenuto di C in D, invece di uno solo? Trasferisce il contenuto di C in TMP, e poi il contenuto di TMP in D. Non sarebbe più semplice trasferire il contenuto di C in D direttamente facendo uso di uno stato singolo?*

Risposta: Questo non è possibile a causa della struttura scelta per i registri interni. Infatti, tutti i registri interni sono parte di una singola RAM, una memoria di lettura e scrittura interna al chip del microprocessore. Solo una parola alla volta può essere indirizzata o scelta entro una RAM (a singola porta). Per questa ragione non è possibile leggere e scrivere in, o da, una RAM in due posizioni differenti. Sono richiesti due cicli di RAM. Diventa necessario per prima cosa leggere i dati dal registro della RAM, e immagazzinarli in un registro temporaneo TMP, poi, riscriverli nel registro destinazione finale, che qui è D. Questo è un progetto insufficiente ed inadeguato.

Comunque questa limitazione è comune virtualmente a tutti i microprocessori monolitici. Per risolvere il problema ci sarebbe bisogno di una RAM a doppia porta.

Questa limitazione non è intrinseca ai microprocessori e normalmente non esiste nel caso dei dispositivi del tipo a bit-slice (che operano su più parole contemporaneamente). È un risultato della costante ricerca nella direzione della densità logica del chip e può essere eliminato nel futuro.

ESERCIZIO IMPORTANTE:

A questo punto è raccomandato che il lettore riesamini da solo la sequenza di questa semplice istruzione prima che procediamo ad istruzioni più complesse. Per questo scopo, andate indietro alla Figura 2-14. Mettete insieme alcuni simboli di piccole dimensioni come fiammiferi, graffette, ecc. Poi spostate i simboli sulla Figura 2-14 per simulare il flusso dei dati dai registri nei bus. Per esempio, depositate un simbolo nel PC. T1 farà uscire il simbolo contenuto nel PC sul bus degli indirizzi verso la memoria. Continuate l'esecuzione simulata in questo modo finchè vi sentirete a vostro agio con i trasferimenti lungo i bus e tra i registri. A questo punto, dovreste essere in grado di andare avanti. Ora saranno studiate istruzioni progressivamente più complesse:

ADD A, r

Questa istruzione significa: "Somma il contenuto del registro r (specificato da un codice binario SSS) all'accumulatore (A), e deposita il risultato nell'accumulatore". Questa è una istruzione *implicita*.

È chiamata implicita poichè non fa esplicitamente riferimento ad un secondo registro. L'istruzione si riferisce esplicitamente al registro r. Questo sottintende che l'altro registro implicato nell'operazione è l'accumulatore. L'accumulatore, quando è usato in una tale istruzione implicita, è riferito sia come sorgente che come destinazione. I dati saranno depositati nell'accumulatore come risultato di questa addizione. Il vantaggio di tale istruzione implicita è quello che il suo codice operativo è lungo soltanto otto bit. Richiede solo un campo di registro da tre bit per la descrizione dettagliata di r. Questo è un modo rapido per eseguire una operazione di addizione.

Esistono nel sistema altre istruzioni implicite che faranno riferimento ad altri registri specializzati.

Esempi più complessi di tali istruzioni implicite sono, per esempio, le operazioni PUSH e POP, le quali trasferiranno le informazioni tra la sommità dello stack e l'accumulatore, e allo stesso tempo aggiorneranno il puntatore dello stack (SP) decrementandolo o incrementandolo. Implicitamente manipolano il registro SP.

L'esecuzione dell'istruzione ADD A, r sarà ora esaminata dettagliatamente. Questa istruzione richiederà due cicli di macchina, M1 e M2. Come sempre, durante i primi tre stati di M1, l'istruzione è prelevata dalla memoria e depositata nel registro IR. All'inizio di T4, è decodificata e può essere eseguita. Qui sarà supposto che il registro B sia addizionato all'accumulatore. Allora, il codice per l'istruzione sarà: 10000000 (il codice per il registro B è 000). Il codice equivalente per l'8080 è ADD r.

T4: (SSS) → TMP, (A) → ACT

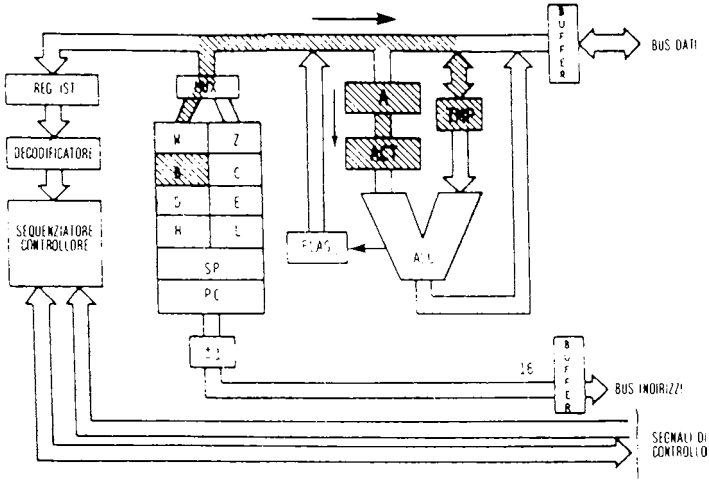


Figura 2-23: Avvengono simultaneamente 2 trasferimenti

Saranno eseguiti simultaneamente due trasferimenti. Primo, il contenuto del registro sorgente specificato (qui B) è trasferito in TMP, cioè, all'ingresso destro dell'unità ALU (vedere Figura 2-23). Allo stesso tempo, il contenuto dell'accumulatore è trasferito nell'accumulatore temporaneo (ACT). Ispezionando la Figura 2-23, constaterete che questi trasferimenti possono avvenire in parallelo. Infatti essi usano percorsi diversi all'interno del sistema.

Il trasferimento da B a TMP usa il bus dei dati interno. Il trasferimento dall'ACT usa un breve percorso interno indipendente da questo bus dei dati. Per guadagnare tempo, entrambi i trasferimenti sono fatti simultaneamente. A questo punto, sia l'ingresso destro che sinistro dell'ALU sono correttamente condizionati. L'input sinistro dell'ALU è ora condizionato dal contenuto dell'accumulatore, e l'input destro dell'ALU è condizionato dal contenuto del registro B. Siamo pronti ad eseguire l'addizione. Normalmente ci aspetteremmo di vedere che l'addizione avvenga durante lo stato T5 di M1. Invece questo stato non è semplicemente usato. L'addizione non è eseguita! Entreremo nel ciclo di macchina M2. Durante lo stato T1 non avviene niente! È solo nello stato T2 di M2 che avviene l'addizione (riferitevi ad ADD r nella Figura 2-27):

T2 di M2: (ACT) + (TMP) → A

Il contenuto di ACT viene sommato al contenuto di TMP, e il risultato è infine depositato nell'accumulatore. Vedere la Figura 2-24. L'operazione è ora completa.

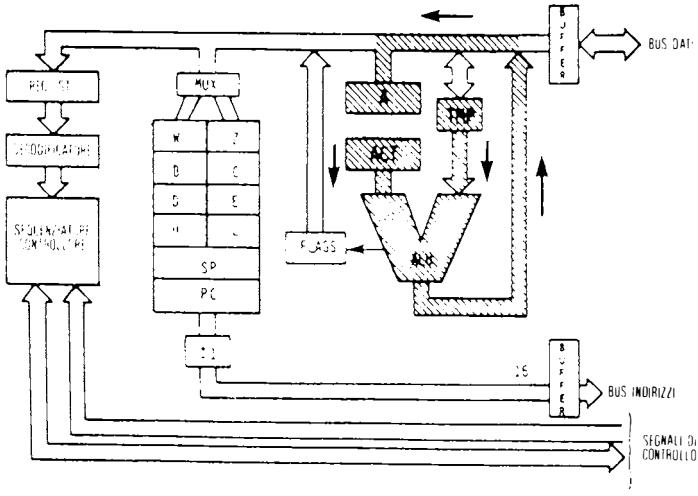


Figura 2-24: Conclusione di ADDr

Domanda: *Perchè il completamento dell'addizione è stato rimandato fino allo stato T2 del ciclo di macchina M2, piuttosto che eseguito durante lo stato T5 di M1?* (Questa è una domanda difficile, che richiede una comprensione del progetto della CPU. Comunque la tecnica ora usata è fondamentale nel progetto della CPU sincronizzata al clock. Provate a vedere cosa succede.)

Risposta: Questo è un "trucco" di progetto standard usato nella maggior parte delle CPU. È chiamata "fetch/execute overlap" (sovrapposizione delle fasi di fetch e di esecuzione). L'idea fondamentale è la seguente: riguardando la figura 2-23 può essere visto che la vera esecuzione dell'addizione richiederà soltanto l'uso dell'ALU e del bus dei dati. In particolare, non si accederà al registro RAM (blocco dei registri). Noi (oppure l'unità di controllo) sappiamo che i prossimi tre stati che saranno eseguiti dopo il comportamento di ogni istruzione saranno T1, T2, T3 del ciclo di macchina M1 della istruzione seguente. Riguardando all'esecuzione di questi tre stati, si può vedere che la loro esecuzione richiederà soltanto l'accesso al contatore di programma (PC) e l'uso del bus degli indirizzi. L'accesso al contatore di programma (program counter) richiederà l'accesso al registro RAM. (Questo spiega perché lo stesso trucco non ha potuto essere usato nella istruzione LD r, r'). Perciò è possibile usare simultaneamente l'area ombreggiata nella Figura 2-17 e l'area ombreggiata nella Figura 2-24.

Il bus dei dati viene usato durante lo stato T1 di M1 per portare fuori le informazioni di stato. Non può essere usato per l'addizione che desideriamo eseguire. Per questo motivo, diventa necessario aspettare fino allo stato T2 prima che l'addizione possa essere effettivamente eseguita. Questo è quello che è accaduto nel grafico: l'addizione è completata durante lo stato T2 di M2. Il meccanismo è ora spiegato. Il vantaggio di questo approccio ora dovrebbe essere chiaro. Presumiamo di avere realizzato lo schema più semplice, ed eseguito l'addizione durante lo stato T5 del ciclo di macchina M1.

La durata dell'istruzione ADD sarebbe stata $5 \times 500 = 2500$ ns. Se invece realizziamo il nuovo metodo di sovrapposizione (overlap approach), una volta eseguito lo stato T4, inizia la fase di «fetch» della nuova istruzione. In una maniera che è invisibile a questa istruzione seguente l'unità di controllo "intelligente" userà lo stato T2 per eseguire la fine dell'addizione. Sul grafico T2 è mostrato come parte di M2. Concettualmente, M2 sarà il secondo ciclo di macchina dell'addizione. Infatti, questo M2 sarà sovrapposto, cioè, sarà identico al ciclo di macchina M1 dell'istruzione seguente. Per il programmatore, il ritardo introdotto tramite l'istruzione ADD

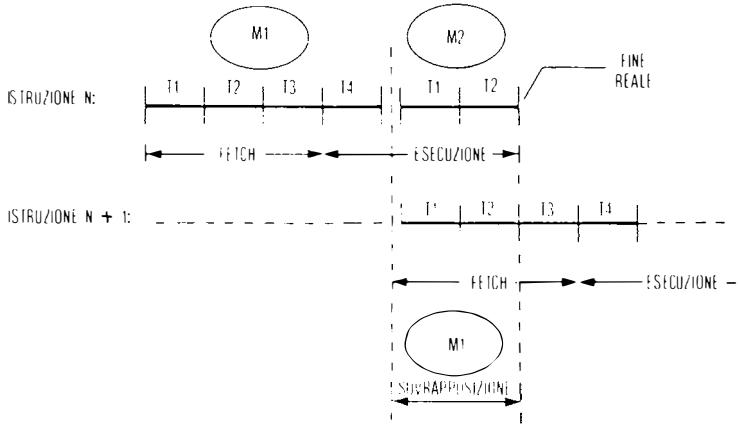
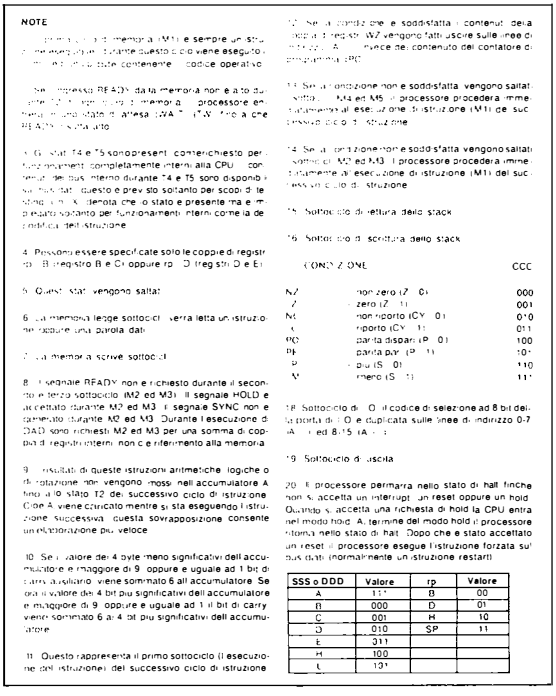


Figura 2-25: Sovrapposizione dei cicli di FETCH e ESECUZIONE durante T1 e T2

sarà solo di quattro stati, cioè, 4 x 500 = 2000 ns, invece di 2500 ns che risultavano dallo schema più semplice. Il miglioramento di velocità è di 500 ns, o 20%!

La tecnica di "overlap" è illustrata sulla Figura 2-25. È usata ogni qualvolta possibile per aumentare la velocità di esecuzione apparente del microprocessore. Naturalmente, non è possibile sovrapporre in tutti i casi. I bus necessari e in generale le "facilities" devono essere disponibili senza conflitto. L'unità di controllo "sa" se è possibile un overlap.



NOTE

[illegible] memoria (M1) è sempre un stru[illegible] durante questo ciclo viene eseguito [illegible] contenente il codice operativo.

[illegible] processo READY data la memoria non è alto du[illegible] processore en[illegible] READY [illegible] attesa [illegible] che READY [illegible] alto.

[illegible] stati T4 e T5 sono presenti come richiesto per [illegible] completamente interni alla CPU con [illegible] durante T4 e T5 sono disponibili [illegible] questo è previsto soltanto per scopi di te[illegible] denota che lo stato è presente ma è im[illegible] soltanto per funzionamenti interni come la de[illegible] dell'istruzione.

4. Possono essere specificate solo le coppie di registri rp: B registro B e C oppure rp: D (registri D e E).

5. Questi stati vengono saltati.

6. La memoria legge sottociclo: verrà letta un'istruzione oppure una parola dati.

7. La memoria scrive sottociclo.

8. Il segnale READY non è richiesto durante il secondo e terzo sottociclo (M2 ed M3). Il segnale HOLD è accettato durante M2 ed M3. Il segnale SYNC non è richiesto durante M2 ed M3. Durante l'esecuzione di DAD sono richiesti M2 ed M3 per una somma di coppia di registri interni: non c'è riferimento alla memoria.

9. I risultati di queste istruzioni aritmetiche, logiche o di rotazione non vengono messi nell'accumulatore A fino al 0 stato T2 del successivo ciclo di istruzione. Come A viene cancellato mentre si sta eseguendo l'istruzione successiva, questa sovrapposizione consente un'elaborazione più veloce.

10. Se il valore dei 4 bytes meno significativi dell'accumulatore è maggiore di 9 oppure è uguale ad 1 bit di carry ausiliario viene sommato 6 all'accumulatore. Se ora il valore dei 4 bit più significativi dell'accumulatore è maggiore di 9 oppure è uguale ad 1 il bit di carry viene sommato 6 ai 4 bit più significativi dell'accumulatore.

11. Questo rappresenta il primo sottociclo (l'esecuzione dell'istruzione) del successivo ciclo di istruzione.

12. Se la condizione è soddisfatta i contenuti della coppia di registri WZ vengono fatti uscire sulle linee di [illegible] A invece del contenuto del contatore di programma (PC).

13. Se la condizione non è soddisfatta vengono saltati sottocicli M4 ed M5. Il processore procederà immediatamente all'esecuzione di istruzione (M1) del successivo ciclo di istruzione.

14. Se la condizione non è soddisfatta vengono saltati sottocicli M2 ed M3. Il processore procederà immediatamente all'esecuzione di istruzione (M1) del successivo ciclo di istruzione.

15. Sottociclo di lettura dello stack.

16. Sottociclo di scrittura dello stack.

CONDIZIONE		CCC
NZ	non zero (Z = 0)	000
Z	zero (Z = 1)	001
NC	non riporto (CY = 0)	010
C	riporto (CY = 1)	011
PO	parità dispari (P = 0)	100
PE	parità pari (P = 1)	101
P	più (S = 0)	110
M	meno (S = 1)	111

18. Sottociclo di I/O: il codice di selezione ad 8 bit della porta di I/O è duplicato sulle linee di indirizzo 0-7 (A[illegible]) ed 8-15 (A[illegible]).

19. Sottociclo di uscita.

20. Il processore permarrà nello stato di hall finché non si accetta un interrupt, un reset oppure un hold. Quando si accetta una richiesta di hold la CPU entra nel modo hold. A termine del modo hold il processore ritorna nello stato di hall. Dopo che è stato accettato un reset il processore esegue l'istruzione forzata sul bus dati (normalmente un'istruzione restart).

SSS o DDD	Valore	rp	Valore
A	111	B	00
B	000	D	01
C	001	H	10
D	010	SP	11
E	011		
H	100		
L	101		

Figura 2-26: Abbreviazioni della Intel

MNEMONIC	MACHINE OPERATION	M1					M2		
	$D_7D_6D_5D_4$ $D_3D_2D_1D_0$	T1	T2,2	T3	T4	T5	T1	T2,2	T3
MVI r	0 0 1 1 0	PC OUT STATUS	PC ← PC + 1	INST → TMP/IR	[PC]→TMP	TMP→ACT			
MVI M	0 0 1 1 0				[PC]→TMP		HL OUT STATUS[6]	DATA	→ TMP
MVI M	0 0 1 1 0				[PC]→TMP		HL OUT STATUS[6]	TMP	→ DATA BUS
SPHL	1 1 1 1 1				[HL]	→ SP			
MVI data	0 0 1 1 0				X		PC OUT STATUS[6]	B2	→ TMP
MVI M data	0 0 1 1 0				X			B2	→ TMP
LXI data	0 0 R P 0 1				X			PC ← PC + 1	B3 → Z
LDA addr	0 0 1 1 1 0				X			PC ← PC + 1	B2 → Z
STA addr	0 0 1 1 1 0				X			PC ← PC + 1	B2 → Z
SHLD addr	0 0 1 0 1 0				X			PC ← PC + 1	B2 → Z
SHLD addr	0 0 1 0 1 0				X		PC OUT STATUS[6]	PC ← PC + 1	B2 → Z
LDAX rp[4]	0 0 R P 1 0 1 0				X		rp OUT STATUS[6]	DATA	→ A
STAX rp[4]	0 0 R P 0 0 1 0				X		rp OUT STATUS[6]	A	→ DATA BUS
XCHG	1 1 1 0 1 1 1 1				[HL]→ACT				
ADD r	1 0 0 0 0 S S S				[SSS]→TMP [A]→ACT		[9]	[ACT]+[TMP]→A	
ADD M	1 0 0 0 0 1 1 0				[A]→ACT		HL OUT STATUS[6]	DATA	→ TMP
ADI data	1 1 0 0 0 1 1 0				[A]→ACT		PC OUT STATUS[6]	PC ← PC + 1	B2 → TMP
ADC r	1 0 0 0 1 S S S				[SSS]→TMP [A]→ACT		[9]	[ACT]+[TMP]+CY→A	
ADC M	1 0 0 0 1 1 1 0				[A]→ACT		HL OUT STATUS[6]	DATA	→ TMP
ACI data	1 1 0 0 1 1 1 0				[A]→ACT		PC OUT STATUS[6]	PC ← PC + 1	B2 → TMP
SUB r	1 0 0 1 0 S S S				[SSS]→TMP [A]→ACT		[9]	[ACT]−[TMP]→A	
SUB M	1 0 0 1 0 1 1 0				[A]→ACT		HL OUT STATUS[6]	DATA	→ TMP
SUI data	1 1 0 1 0 1 1 0				[A]→ACT		PC OUT STATUS[6]	PC ← PC + 1	B2 → TMP
SBB r	1 0 0 1 1 S S S				[SSS]→TMP [A]→ACT		[9]	[ACT]−[TMP]−CY→A	
SBB M	1 0 0 1 1 1 1 0				[A]→ACT		HL OUT STATUS[6]	DATA	→ TMP
SBI data	1 1 0 1 1 1 1 0				[A]→ACT		PC OUT STATUS[6]	PC ← PC + 1	B2 → TMP
INR r	0 0 D D 0 1 0 0				[DDD]→TMP [TMP]+1→ALU	ALU→DDD			
INR M	0 0 1 1 0 1 0 0				X		HL OUT STATUS[6]	DATA [TMP]+1	→ TMP → ALU
DCR r	0 0 D D 0 1 0 1				[DDD]→TMP [TMP]+1→ALU	ALU→DDD			
DCR M	0 0 1 1 0 1 0 1				X		HL OUT STATUS[6]	DATA [TMP]−1	→ TMP → ALU
INX rp	0 0 R P 0 0 1 1				[RP] + 1	→ RP			
DCX rp	0 0 R P 1 0 1 1				[RP] − 1	→ RP			
DAD rp[8]	0 0 R P 1 0 0 1				X		[H]→ACT	[L]→TMP [ACT]+[TMP]→ALU	ALU→L CY
DAA	0 0 1 0 0 1 1 1				DAA→A FLAGS[10]				
ANA r	1 0 1 0 0 S S S				[SSS]→TMP [A]→ACT		[9]	[ACT]+[TMP]→A	
ANA M	1 0 1 0 0 1 1 0	PC OUT STATUS	PC ← PC + 1	INST → TMP/IR	[A]→ACT		HL OUT STATUS[6]	DATA	→ TMP

Figura 2-27: Formati delle istruzioni Intel

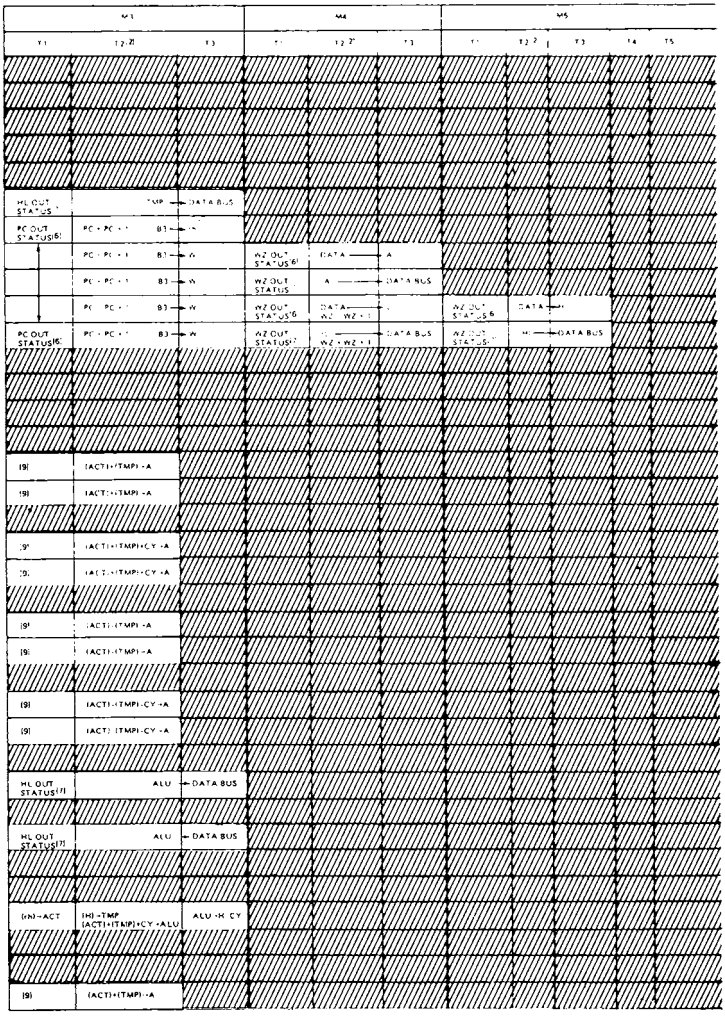


Figura 2-27: Formati delle istruzioni Intel (segue)

MNEMONICO	CODICE OPERATIVO		M1					M2		
	$D_7D_6D_5D_4$	$D_3D_2D_1D_0$	T1	T2	T3	T4	T5	T1	T2	T3
[illegible]	[illegible]	[illegible]	PC OUT STATUS	PC = PC + 1	INST = TMP/IR	[illegible]		PC OUT STATUS[6]	PC = PC + 1	B2 ← TMP
[illegible]	[illegible]	[illegible]				[illegible]		[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]				[illegible]		PC OUT STATUS[6]		DATA ← TMP
[illegible]	[illegible]	[illegible]		

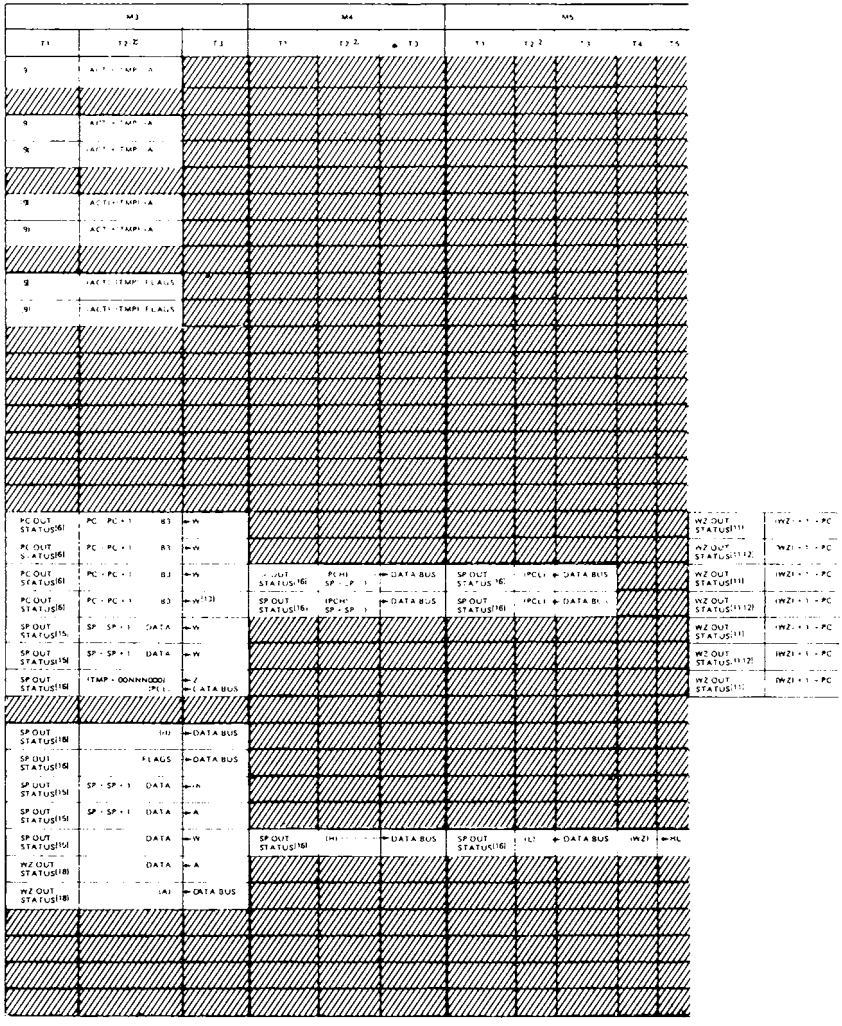


Figura 2-27: Formati delle istruzioni Intel (segue)

Domanda: *Sarebbe possibile andare oltre usando questo schema e usare anche lo stato T3 di M2 se dobbiamo eseguire una istruzione più lunga?*

Per chiarire il meccanismo di sequenzializzazione interna, si consiglia di esaminare la Figura 2-27 che mostra la dettagliata esecuzione delle istruzioni per l'8080. Lo Z80 include tutte le istruzioni dell'8080, e in più altre. Le informazioni presentate nella Figura 2-27 non sono utilizzabili per lo Z80. Qui è mostrato per il suo valore educativo nella comprensione del modo di funzionare di questo microprocessore. L'equivalenza tra le istruzioni dello Z80 e dell'8080 è mostrata nell'Appendice.

Ora sarà esaminata una istruzione più complessa.

ADD A, (HL)

Il codice operativo per questa istruzione è 10000110.

Questa istruzione significa "somma all'accumulatore il contenuto della locazione di memoria (HL)". La locazione di memoria è specificata tramite un sistema piuttosto strano. È la locazione di memoria il cui indirizzo è contenuto nei registri H ed L. Questa istruzione presuppone che questi due speciali registri (HL) siano stati caricati con i contenuti prima di eseguire l'istruzione. Il contenuto a 16 bit di questi registri ora specificherà l'indirizzo nella memoria dove risiedono i dati. Questi dati saranno sommati all'accumulatore, ed il risultato sarà lasciato nell'accumulatore.

Questa istruzione ha una storia. È stata fornita per dare compatibilità tra il vecchio 8008, e il suo successore, lo 8080. L'antecedente 8008 non era fornito di una capacità di indirizzamento diretto della memoria! La procedura usata per accedere al contenuto della memoria era quella di caricare i due registri H ed L, e poi eseguire una istruzione riferendosi ad H e L. ADD A, (HL) è proprio una di quelle istruzioni. Deve essere sottolineato che l'8080 e lo Z80 non sono limitati nella stessa maniera dell'8008 nella capacità di indirizzamento diretto della memoria. Essi hanno l'indirizzamento diretto della memoria. La possibilità di usare i registri H ed L diventa un vantaggio aggiunto, non uno svantaggio, come era nel caso dell'8008.

Ora seguiamo l'esecuzione di questa istruzione (è chiamata ADDM per l'8080 ed è la 16-esima istruzione in Figura 2-27). Gli stati T1, T2 e T3 di M1 saranno usati, come sempre, per

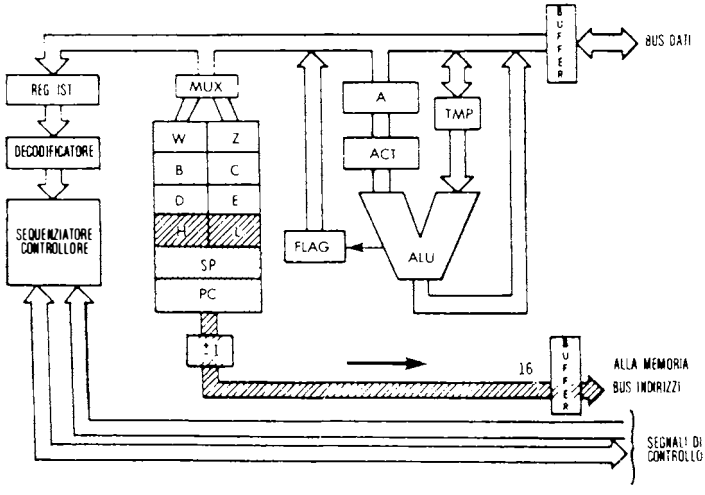


Figura 2-28: Trasferimento del contenuto di HL nel Bus degli indirizzi

prelevare l'istruzione. Durante lo stato T4, il contenuto dell'accumulatore è trasferito al suo buffer, ACT, e l'input sinistro dell'ALU è condizionato.

Si deve poi accedere alla memoria per fornire il secondo byte di dati che sarà sommato all'accumulatore. L'indirizzo di questo byte di dati è contenuto in H ed L. Il contenuto di H ed L dovrà perciò essere trasferito sul bus degli indirizzi, da dove saranno portati alla memoria. Facciamolo.

Durante il ciclo di macchina M2, leggiamo: HL OUT. H ed L sono depositati sul bus degli indirizzi, nello stesso modo in cui il PC era generalmente depositato nelle istruzioni precedenti. Come osservazione è già stato indicato che durante T1 lo *stato* è in uscita sul bus dei dati, ma qui non se ne farà nessun uso. Da un punto di vista semplificato, si richiederanno due stati: uno perché la memoria legga i suoi dati, e uno perchè i dati diventino utilizzabili e trasferiti sull'input destro dell'ALU, TMP.

Entrambi gli input (ingressi) dell'ALU sono ora condizionati. La situazione è analoga a quella in cui eravamo con la precedente istruzione ADD A, r: entrambi gli input dell'ALU sono condizionati. Noi dobbiamo semplicemente sommare (ADD) come prima. Sarà usata come in precedenza una tecnica di fetch execute overlap, e, invece di eseguire l'addizione entro lo stato T4 di M2, l'esecuzione finale è ritardata fino allo stato T2 di M3. Nella Figura 2-27 si può vedere che durante T2 abbiamo veramente: ACT + TMP A. L'addizione è finalmente compiuta, il contenuto di ACT è aggiunto a TMP ed il risultato è depositato nell'accumulatore A.

Domanda: *Qual è il tempo di esecuzione apparente (al programmatore) per questa istruzione? È 7.5 µs, oppure è 4.5 µs?*

Ora sarà esaminata un'altra istruzione più complessa che è una istruzione ad indirizzamento diretto della memoria che usa due registri invisibili W e Z:

LD A, (nn)

L'opcode (codice operativo) è 00111010. L'equivalente per l'8080 è LDA indirizzo. Come sempre, gli stati T1, T2 e T3 di M1 saranno usati per prelevare l'istruzione dalla memoria. Il T4 è usato, ma non può essere descritto nessun risultato visibile. Durante lo stato T4, l'istruzione è infatti decodificata. L'unità di controllo allora scopre che deve prelevare i due byte successivi di questa istruzione per ottenere l'indirizzo dal quale l'accumulatore sarà caricato. L'effetto di questa istruzione è di caricare l'accumulatore dal contenuto della memoria il cui indirizzo è specificato nei byte 2 e 3 dell'istruzione. Notate che lo stato T4 è necessario per *decodificare* l'istruzione. Potrebbe essere considerato uno spreco di tempo dal momento in cui è necessaria solo una parte dello stato per fare la decodifica. È così. Comunque, questa è la filosofia del cosiddetto "*clock-synchronons logic*". Poichè le *microistruzioni* sono usate internamente per compiere la decodificazione e l'esecuzione, questa è la penale che deve essere pagata in cambio dei vantaggi della microprogrammazione. La struttura di questa istruzione compare nella Figura 2-29.

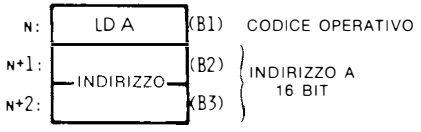


Figura 2-29: LD A, (ADDRESS) È una istruzione di tre parole

Ora saranno prelevati i due byte dopo l'istruzione. Specificheranno un indirizzo (vedere la Figura 2-30).

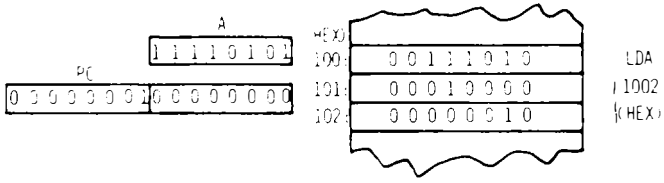


Figura 2-30: Prima dell'esecuzione di LDA

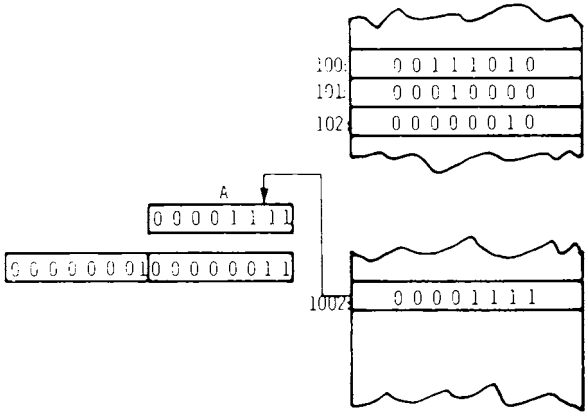


Figura 2-31: Dopo l'esecuzione di LDA

L'effetto della istruzione è mostrato nelle Figure 2-30 e 2-31.

Due registri speciali sono disponibili all'unità di controllo all'interno dello Z80 (ma non al programmatore). Sono "W" e "Z", e sono mostrati nella Figura 2-28.

Secondo Ciclo di Macchina M2: Come al solito, i primi due stati, T1 e T2, sono usati per prelevare il contenuto della posizione di memoria indicata nel PC. Durante T2, il program counter, PC, è incrementato. A volte verso la fine di T2, i dati diventano disponibili dalla memoria, e appaiono sul bus dei dati. Verso la fine di T3, la parola che è stata prelevata dall'indirizzo di memoria PC (B2, il secondo byte dell'istruzione) è disponibile sul bus dei dati. Ora deve essere immagazzinato in un registro temporaneo. È depositato nello Z: B2 → Z (vedere la Figura 2-32).

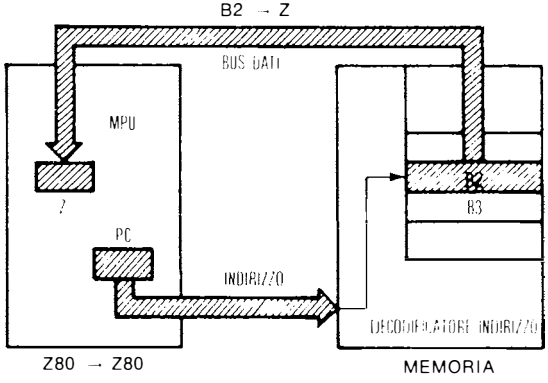


Figura 2-32: Il secondo byte dell'istruzione va in Z

Ciclo di Macchina M3: Di nuovo, il PC è depositato sul bus degli indirizzi, incrementato, ed infine il terzo byte, B3, è letto dalla memoria e depositato nel registro W del microprocessore. A questo punto, cioè, alla fine dello stato T3 di M3, i registri W e Z all'interno del microprocessore contengono B2 e B3, cioè, l'indirizzo a 16 bit completo che era originariamente contenuto nelle due parole che seguivano l'istruzione nella memoria. Ora può essere completata l'esecuzione. W e Z contengono un indirizzo. Questo indirizzo dovrà essere inviato alla memoria, per estrarre i dati.

Questo è fatto nel prossimo ciclo di memoria:

Ciclo di Macchina M4: Questa volta, W e Z sono inviati sul bus degli indirizzi. L'indirizzo a 16 bit è inviato alla memoria, e per la fine dello stato T2, diventano utilizzabili i dati che corrispondono al contenuto della posizione di memoria specificata. Questi sono infine depositati in A alla fine dello stato T3. Cosi termina l'esecuzione di questa istruzione.

Questo illustra l'uso di una istruzione immediata. Questa istruzione richiede tre byte per immagazzinare un *indirizzo esplicito* da due byte. Questa istruzione richiede anche quattro cicli di memoria, poichè ha richiesto di andare in memoria tre volte per estrarre i tre byte di questa istruzione da tre parole, più un altro accesso in memoria per prelevare i dati specificati dall'indirizzo. È una istruzione lunga. Comunque, è anche una istruzione fondamentale per caricare l'accumulatore con contenuti specificati che risiedono in una posizione di memoria nota. Può essere notato che questa istruzione richiede l'uso dei registri W e Z.

Domanda: *Questa istruzione potrebbe aver usato registri diversi da W, Z all'interno del sistema?*

Risposta: No. Se questa istruzione avesse usato altri registri, per esempio i registri H e L, avrebbe modificato il loro contenuto. Dopo l'esecuzione di questa istruzione, il contenuto di H e L sarebbe stato perduto. In un programma è sempre presupposto che un'istruzione non modificherà nessun registro se non quelli che sta implicitamente usando. Una istruzione che carica l'accumulatore non dovrebbe distruggere il contenuto di nessun altro registro. Per questa ragione, diventa necessario fornire i due registri extra, W e Z, per uso interno dell'unità di controllo.

Domanda: *Sarebbe possibile usare il PC invece di W e Z?*

Risposta: Decisamente no. Sarebbe disastroso. Il lettore dovrebbe analizzare questo.

Ora sarà studiato un altro tipo di istruzione: una istruzione di diramazione o salto (branch o jump), la quale modifica la sequenza in cui le istruzioni sono eseguite all'interno del programma. Fin'ora, abbiamo presupposto che le istruzioni fossero eseguite sequenzialmente.

Esistono istruzioni che permettono al programmatore di saltare dalla sequenza ad un'altra istruzione all'interno del programma, o in termini pratici, di saltare ad un'altra area della memoria che contiene il programma, oppure ad un altro indirizzo. Una tale istruzione è:

JP nn

Questa istruzione appare alla Linea 18 della Figura 2-27 come "JMP addr". La sua esecuzione sarà descritta seguendo la linea orizzontale della Tabella. Questa è di nuovo una istruzione di tre parole. La prima parola è l'opcode (codice operativo), e contiene 11000011. Le due parole che seguono contengono l'indirizzo a 16 bit al quale sarà fatto il salto. Concettualmente, l'effetto di questa istruzione è di sostituire il contenuto del program counter con i 16 bit che seguono il codice operativo "JUMP".

In pratica, per ragioni di efficienza, sarà compiuto un approccio un po' differente.

Come prima, i primi tre stati di M1 corrispondono alla istruzione di prelievo. Durante lo stato T4 l'istruzione è decodificata e non è registrato nessun altro evento (X).

I due cicli di macchina successivi sono usati per prelevare i byte B2 e B3 dell'istruzione. Du-

rante M2, B2 è prelevato e depositato nel registro interno W. Le due fasi successive saranno compiute dal processore durante il prossimo prelievo di istruzione. come è già stato il caso con l'addizione. Esse saranno eseguite invece delle usuali fasi per T1 e T2 della istruzione seguente. Guardiamole.

Le due fasi successive saranno: WZ OUT e (WZ) + 1 · PC. In altre parole, il contenuto di WZ sarà usato invece del contenuto del PC durante il successivo prelievo dell'istruzione. L'unità di controllo avrà registrato il fatto che stava per essere eseguito un jump ed eseguirà in modo differente l'inizio dell'istruzione dopo.

L'effetto di questi due stati extra è il seguente:

L'indirizzo posto sul bus degli indirizzi del sistema, sarà l'indirizzo contenuto in W e Z. In altre parole, l'istruzione successiva sarà prelevata dall'indirizzo che era contenuto in W e Z. Ciò è effettivamente un *jump*. In aggiunta, il contenuto di WZ sarà aumentato per 1 e depositato nel program counter, in modo che l'istruzione successiva sia prelevata correttamente usando come sempre il PC.

L'effetto è perciò corretto.

Domanda: *Perchè non abbiamo caricato direttamente il contenuto del PC? Perchè usiamo i registri intermedi W e Z?*

Risposta: Non è possibile usare il PC. Se avessimo caricato la parte più bassa del PC (PCL) con B2 invece di usare Z, avremmo distrutto il PC! Sarebbe allora diventato impossibile prelevare B3.

Domanda: *Sarebbe possibile usare solo Z, invece di W e Z?*

Risposta: Sì, ma sarebbe più lento. Si sarebbe potuto caricare Z con B2, poi prelevare B3 e depositarlo nella metà alta del PC (PCH). Comunque, poi sarebbe diventato necessario trasferire Z nel PCL, prima di usare il contenuto del PC. Ciò rallenterebbe il processo. Per questa ragione, dovrebbero essere usati sia W che Z. Inoltre, e per risparmiare del tempo, W e Z non sono trasferiti nel PC. Sono direttamente forniti al bus degli indirizzi per prelevare l'istruzione successiva. Comprendere questo punto è cruciale per la comprensione dell'efficiente esecuzione delle istruzioni all'interno del microprocessore.

Domanda: *(Solo per il lettore attento ed informato). Cosa succede nel caso di un interrupt alla fine di M3? (Se l'esecuzione dell'istruzione è sospesa a questo punto. il contatore di programma punta all'istruzione che segue il jump, e sarà perduto l'indirizzo di jump contenuto in W e Z).*

La risposta è lasciata come un interessante esercizio per il lettore attento.

Le descrizioni dettagliate che abbiamo presentato per l'esecuzione di istruzioni tipiche dovrebbe chiarire il ruolo dei registri e dei bus interni. Una seconda lettura del paragrafo precedente potrebbe aiutare a guadagnare una comprensione dettagliata del modo di operare internamente dello Z80.

IL CHIP Z80

Per completezza, qui saranno esaminati i segnali del chip del microprocessore Z80. Non è indispensabile capire le funzioni dei segnali dello Z80 per essere in grado di programmarlo.

Il lettore che non è interessato nei dettagli dell'hardware può perciò saltare questo paragrafo. Il pinout (configurazione dei pin di uscita) dello Z80 appare nella Figura 2-33. Sul lato destro dell'illustrazione, il bus degli indirizzi e il bus dei dati eseguono il loro ruolo usuale, come descritto all'inizio di questo capitolo. Qui descriveremo la funzione dei segnali sul bus di controllo. Sono mostrati sulla sinistra della Figura 2-33.

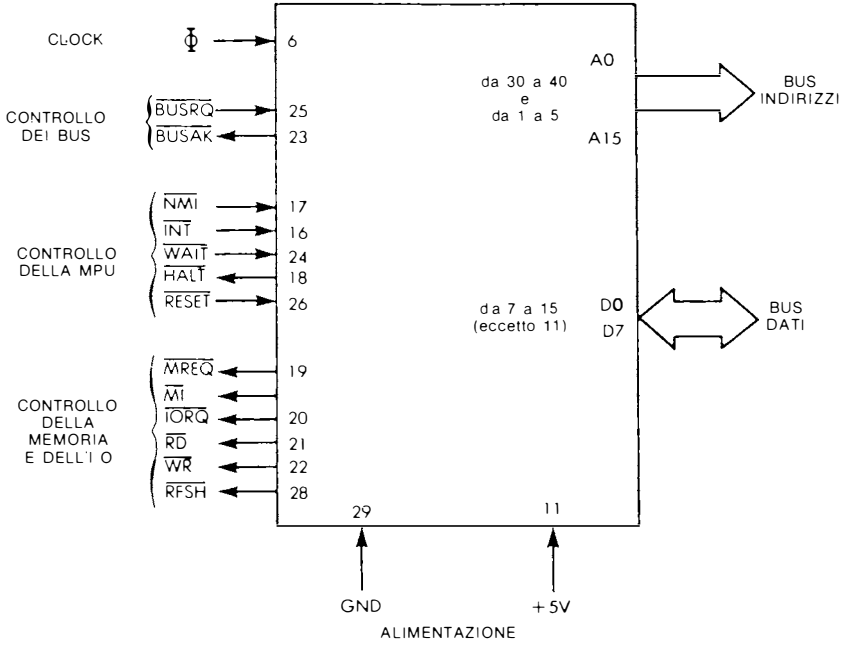


Figura 2-33: Configurazione dei pin di uscita della MPU dello Z80

I segnali di controllo sono stati divisi in quattro gruppi. Saranno descritti, andando dall'alto della Figura 2-33 verso il fondo.

L'input del clock è l'input 0. Lo Z80 incorpora l'oscillatore del clock all'interno del chip del microprocessore. Esternamente è necessario soltanto un resistore di pull-up da 330 ohm. È collegato all'input 0 e a 5 volt. Comunque, a 4 MHz, è richiesto un "clock driver" esterno.

I due segnali di controllo del bus, BUSRQ e BUSAK, sono usati per disinserire lo Z80 dai suoi bus. Essi sono principalmente usati dal DMA, ma potrebbero essere anche usati da un altro processore nel sistema. BUSRQ è il segnale di richiesta di bus. È inviato allo Z80. In risposta, lo Z80 disporrà il suo bus degli indirizzi, il bus dei dati ed i segnali di controllo dell'output a tre stati nello stato di alta impedenza, alla fine del ciclo di macchina corrente. BUSAK è il segnale di riconoscimento emesso dallo Z80 una volta che i bus sono stati posti nello stato di alta impedenza.

Se i segnali di controllo dello Z80 sono messi in relazione al suo stato interno o alla sua sequenziazione:

INT e NMI sono i due segnali di interrupt.

INT è la consueta richiesta d'interrupt.

Gli interrupt saranno descritti nel Capitolo 6. Numerosi dispositivi input/output possono essere collegati alla linea di interrupt INT. Ogni qualvolta una richiesta di interrupt (interrupt request) è presente su questa linea, e quando il FLIP-FLOP che abilita l'interrupt interno è abilitato (IFF), lo Z80 accetterà l'interrupt (purchè il BUSRQ non sia attivo). Allora genererà un segnale di riconoscimento: IORQ (emesso durante lo stato M1). La parte rimanente della sequenza degli eventi è descritta nel capitolo 6. NMI è l'interrupt non schermabile o mascherabile. È sempre accettato dallo Z80, e forza lo Z80 a saltare alla posizione esadecimale 0066. Anch'esso è descritto nel capitolo 6. (È anche presupposto che il BUSRQ non sia attivo). WAIT (attesa) è un segnale usato per sincronizzare lo Z80 con una memoria lenta o dispositivi di im-

put-output. Quando è attivo, questo segnale indica che la memoria o il dispositivo non è ancora pronto per il trasferimento dei dati.

La CPU dello Z80 entrerà allora in uno stato speciale di attesa fino a che il segnale di WAIT diventi inattivo. Quindi riassumerà la normale sequenziazione. HALT è il segnale di riconoscimento fornito dallo Z80 dopo che esso ha eseguito l'istruzione HALT.

In questo stato, lo Z80 aspetta un interrupt esterno e continua a eseguire i NOP per rinfrescare continuamente la memoria.

RESET è il segnale che generalmente inizializza l'MPU, fissa il PC, il registro I e R a "0", disabilita i flip-flop di interrupt e fissa l'"interrupt mode" a "0"; è normalmente usato dopo avere applicato l'alimentazione.

Memoria e controllo di I/O

Dallo Z80 sono generati sei segnali di memoria e di controllo I/O. Sono: il MREQ che è il segnale di richiesta di memoria. Indica che è valido l'indirizzo presente sul bus degli indirizzi. Allora può essere compiuta un'operazione di lettura o scrittura sulla memoria.

MI è il ciclo di macchina 1. Questo ciclo corrisponde al ciclo di fetch di una istruzione.

IORQ è la richiesta di input/output. Indica che è valido l'indirizzo I/O presente sui bit 0-7 del bus degli indirizzi. Allora, può essere eseguita una operazione di lettura o scrittura I/O. L'IORQ è generato anche assieme all'M1 quando lo Z80 riconosce un interrupt. Questa informazione può essere usata dai chip esterni per porre il vettore di risposta dell'interrupt sul bus dei dati. (Le operazioni I/O normali non avvengono mai durante lo stato di M1. La combinazione IORQ più M1 indica una situazione di riconoscimento interrupt). RD è il segnale di lettura della memoria. Indica che lo Z80 è pronto a leggere il contenuto del bus dei dati nel suo accumulatore. Può essere usato da ogni chip esterno, o memoria oppure I/O, per depositare i dati sul bus dei dati.

WR è il segnale di scrittura in memoria. Indica che il bus dei dati trattiene dati validi, pronti per essere scritti nel dispositivo specificato.

RFSH è il segnale di rinfresco (refresh). Quando RFSH è attivo, i sette bit più bassi del bus degli indirizzi contengono un indirizzo di rinfresco per le memorie dinamiche.

Il segnale MREQ è allora usato per eseguire l'aggiornamento tramite la lettura della memoria.

SOMMARIO HARDWARE

Ciò completa la nostra descrizione dell'organizzazione interna dello Z80. I dettagli esatti dell'hardware dello Z80 non sono qui molto importanti. Comunque, il ruolo di ognuno dei registri è importante e dovrebbe essere pienamente compreso prima di passare al capitolo seguente. Ora saranno introdotte le vere istruzioni disponibili sullo Z80, e saranno presentate le tecniche fondamentali di programmazione per lo Z80.

CAPITOLO 3

TECNICHE FONDAMENTALI DI PROGRAMMAZIONE

INTRODUZIONE

Lo scopo di questo capitolo è di presentare le tecniche fondamentali necessarie per scrivere un programma usando lo Z80.

Questo capitolo introdurrà nuovi concetti come il "register management", i "loop" e le subroutine. Metterà a fuoco le tecniche di programmazione usando solo le risorse *interne* dello Z80, cioè, i registri. Saranno sviluppati veri programmi, come i programmi aritmetici. Questi programmi serviranno ad illustrare i vari concetti presentati finora e useranno vere istruzioni. Perciò, sarà visto come possono essere usate le istruzioni per manipolare le informazioni tra la memoria e l'MPU, così come per manipolare le informazioni all'interno della stessa MPU. Il prossimo capitolo tratterà poi dettagliatamente le istruzioni utilizzabili sullo Z80. Il capitolo 5 presenterà le Tecniche di Indirizzamento e il capitolo 6 presenterà le tecniche utilizzabili per la manipolazione delle informazioni all'*esterno* dello Z80: Le Tecniche di Input/output.

In questo capitolo, impareremo essenzialmente "facendo".

Esaminando programmi di crescente complessità, impareremo il ruolo delle varie istruzioni, dei registri e applicheremo i concetti sviluppati fino ad ora. Comunque, qui non sarà presentato un importante concetto; è il concetto delle tecniche di indirizzamento. Sarà presentato separatamente nel Capitolo 5 a causa della sua apparente complessità.

Iniziamo immediatamente scrivendo alcuni programmi per lo Z80. Inizieremo con programmi aritmetici. Il "modello del programmatore" di registri dello Z80 è mostrato nella Figura 3-0.

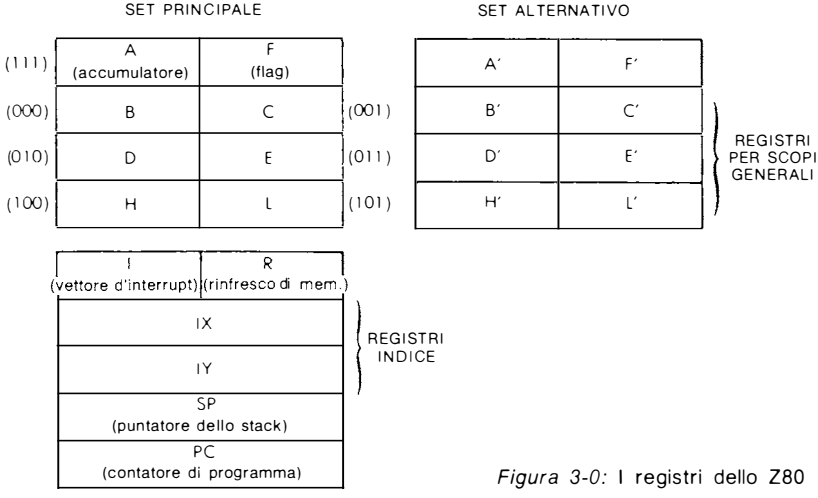


Figura 3-0: I registri dello Z80

PROGRAMMI ARITMETICI

I programmi aritmetici includono l'addizione, la sottrazione, la moltiplicazione e la divisione. I programmi presentati qui opereranno su numeri interi. Questi numeri interi possono essere numeri interi binari positivi o possono essere espressi nella notazione in complemento a due, nel cui caso il bit più a sinistra è il bit di segno (vedere il capitolo 1 per una descrizione della notazione in complemento a due).

L'Addizione ad 8 bit

Sommeremo due operandi ad 8 bit chiamati OP1 e OP2, rispettivamente immagazzinati nell'indirizzo di memoria ADR1 e ADR2. La somma sarà chiamata RES e sarà immagazzinata nell'indirizzo di memoria ADR3. Ciò è illustrato nella Figura 3-1. Il programma che compirà questa addizione è il seguente:

Istruzioni	Commenti
LD A, (ADR1)	CARICA OPR IN A
LD HL, ADR2	CARICA L'INDIRIZZO DI OP2 IN HL
ADD A, (HL)	SOMMA OP2 CON OP1
LD, (ADR3), A	SALVA IL RISULTATO RES IN ADR3

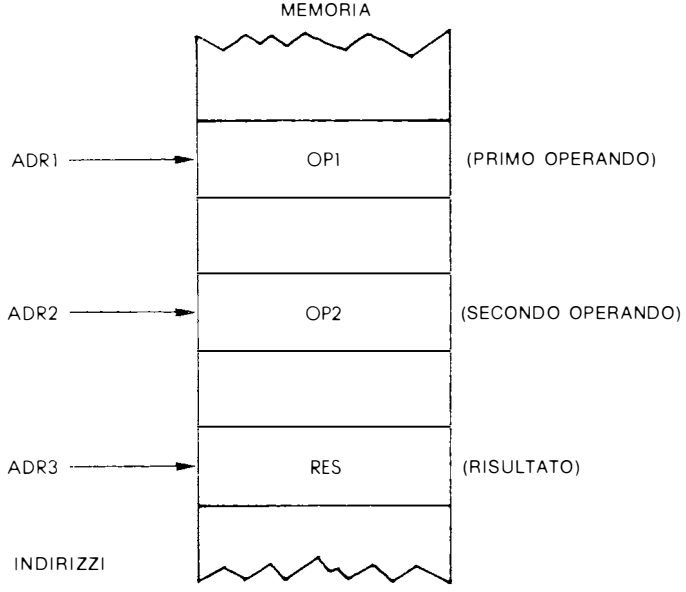


Figura 3-1: Addizione ad 8 bit RES = OP1 + OP2

Questo è il nostro primo programma. Le istruzioni sono elencate alla sinistra ed i commenti appaiono alla destra. Esaminiamo adesso il programma. È un programma a quattro istruzioni.

Ogni linea è chiamata una *istruzione* ed è espressa qui in forma simbolica. Ognuna di queste istruzioni sarà tradotta dal programma *assembler* in uno, due, tre o quattro byte binari. Qui non ci occuperemo della traduzione e guarderemo solo alla rappresentazione simbolica.

La prima linea specifica il caricamento del contenuto di ADR1 nell'accumulatore A. Riferendoci alla Figura 3-1, il contenuto di ADR1 è il primo operando. "OP1". La prima istruzione perciò consiste nel trasferire OP1 dalla memoria all'accumulatore. Ciò è mostrato nella Figura 3-2. "ADR1" è una rappresentazione simbolica per il vero indirizzo a 16 bit nella memoria. Altrove nel programma, sarà definito il simbolo ADR1. Per esempio potrebbe essere definito come essere uguale all'indirizzo "100".

Questa istruzione di "*load*" genererà una *operazione di lettura* dall'indirizzo 100 (vedere la Figura 3-2), il cui contenuto sarà trasferito lungo il bus dei dati e depositato all'interno dell'accumulatore.

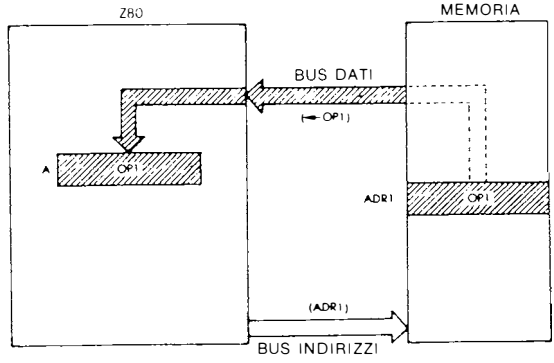


Figura 3-2: LD A, (ADR1): OP1 è caricato dalla memoria

Richiamerete alla mente dal precedente capitolo che le operazioni aritmetiche e logiche operano sull'accumulatore come uno degli operandi "sorgente". (Riferitevi al precedente capitolo per maggiori dettagli). Dal momento in cui vogliamo sommare i due valori OP1 e OP2, dobbiamo per prima cosa caricare OP1 nell'accumulatore. Poi, saremo in grado di sommare il contenuto dell'accumulatore, cioé, sommare OP1 ad OP2.

Il campo più a destra di questa istruzione è chiamato un campo di *commento*. È ignorato dal programma assemblatore all'atto della traduzione, ma è fornito per la leggibilità dei programmi. Per capire cosa fa il programma, è molto importante usare buoni commenti. Ciò è chiamato *documentare* un programma.

Qui il commento è auto-esplicativo: il valore di OP1, che è localizzato all'indirizzo ADR1, è caricato nell'accumulatore A.

Il risultato di questa prima istruzione è illustrato dalla Figura 3-2. La seconda istruzione del nostro programma è:

```
LD HL,   ADR2
```

Specifica: "Carica da (ADR2) nei registri H ed L". Per leggere il secondo operando OP2, dalla memoria, per prima cosa dobbiamo porre il suo indirizzo in una coppia di registri dello Z80, come H ed L. Poi, possiamo sommare il contenuto della posizione di memoria il cui indirizzo è in H ed L, all'accumulatore.

Riferendoci alla Figura 3-1, il contenuto della posizione di memoria ADR2 è OP2, il nostro secondo operando. Il contenuto dell'accumulatore è ora OP1, il nostro primo operando. Come risultato dell'esecuzione di questa istruzione, OP2 sarà prelevato dalla memoria e aggiunto ad OP1. Ciò è illustrato nella Figura 3-3.

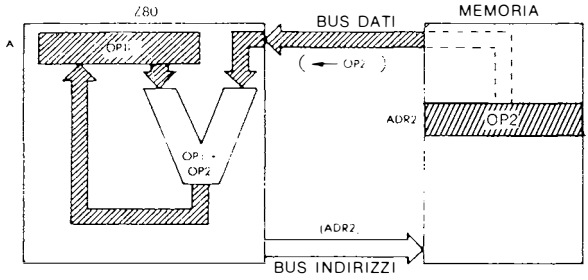


Figura 3-3: ADDA, (HL)

La somma sarà depositata nell'accumulatore. Il lettore ricorderà che, nel caso dello Z80, i risultati delle operazioni aritmetiche sono depositati indietro nell'accumulatore. Negli altri processori, può essere possibile depositare questi risultati in altri registri, o indietro nella memoria.

La somma di OP1 e OP2 è adesso contenuta nell'accumulatore. Per completare il nostro programma, dobbiamo semplicemente trasferire il contenuto dell'accumulatore nella posizione di memoria ADR3, per immagazzinare i risultati alla posizione specificata. Ciò è compiuto dalla quarta istruzione del nostro programma:

LD (ADR3), A

Questa istruzione carica il contenuto di A nell'indirizzo specificato ADR3. L'effetto di questa istruzione finale è illustrato dalla Figura 3-4.

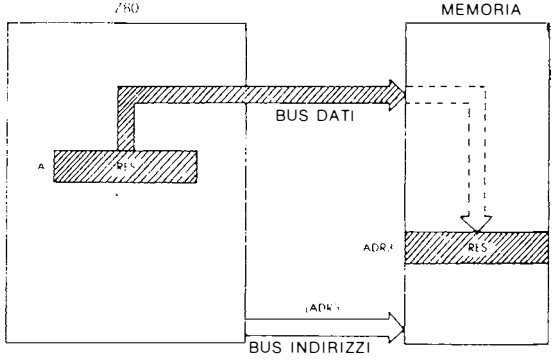


Figura 3-4: LD (ADR3), A (Salva l'Accumulatore nella memoria)

Prima dell'esecuzione dell'operazione ADD, l'accumulatore conteneva OP1 (vedere la Figura 3-3). Dopo l'addizione, è stato scritto un nuovo risultato nell'accumulatore. È "OP1 + OP2". Si ricordi che il contenuto di ogni registro all'interno del microprocessore, così come ogni posizione di memoria, rimane lo stesso dopo che è stata eseguita su questo registro una operazione di lettura. In altre parole, la lettura del contenuto di un registro o di una posizione di memoria non cambia il suo contenuto.

È solo ed esclusivamente, una operazione di *scrittura* in questa posizione di registro che cambierà il suo contenuto. In questo esempio, il contenuto delle posizioni di memoria ADR1 e ADR2 rimangono invariate durante tutto il programma. Comunque, dopo l'istruzione ADD, il

contenuto dell'accumulatore sarà stato modificato, perchè l'output dell'ALU è stato scritto nell'accumulatore. Il precedente contenuto di A non è perduto.

Possono essere usati indirizzi numerici veri invece di ADR1, ADR2 e ADR3. Per conservare indirizzi simbolici, sarà necessario usare le cosiddette "pseudo-istruzioni" le quali specificano il valore di questi indirizzi simbolici, in modo che il programma di assembly possa, durante la traduzione, sostituire gli indirizzi fisici veri. Tali pseudo-istruzioni possono essere, per esempio:

ADR1 = 100 H
ADR2 = 120 H
ADR3 = 200 H

Esercizio 3-1: *Ora chiudete questo libro. Riferitevi solo all'elenco delle istruzioni alla fine del libro. Scrivete un programma per sommare due numeri immagazzinati alle posizioni di memoria LOC1 e LOC2. Depositate i risultati alla posizione di memoria LOC3. Poi, paragonate il vostro programma a quello precedente.*

Addizione a 16 Bit

Una addizione ad 8 bit vi permetterà solo l'addizione di numeri ad 8 bit, cioè, numeri tra 0 e 255, se è usato il binario assoluto. Per applicazioni più pratiche, è necessario aggiungere numeri aventi 16 bit o più, cioè, usare la *precisione multipla*. Qui presenteremo esempi di aritmetica su numeri a 16 bit. Possono essere prontamente estesi a 24, 32 bit o più (sempre multipli di 8 bit). Presumeremo che il primo operando sia immagazzinato nelle posizioni di memoria ADR1 e ADR1-1. Poichè questa volta OP1 è un numero a 16 bit, richiederà due posizioni di memoria di 8 bit. Analogamente, OP2 sarà immagazzinato in ADR2 e ADR2-1. Il risultato deve essere depositato negli indirizzi di memoria ADR3 e ADR3-1. Ciò è illustrato nella Figura 3-5. H indica la metà alta (i bit da 8 a 15), mentre L indica la metà bassa (i bit da 0 a 7).

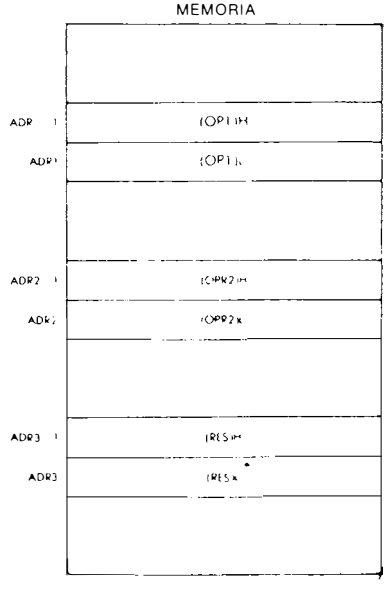


Figura 3-5: Addizione a 16 bit – Gli operandi

La logica del programma è esattamente uguale alla precedente. Per prima cosa, verrà sommata la metà più bassa dei due operandi, poichè il microprocessore può sommare solo 8 bit alla volta. Ogni riporto generato dall'addizione di questi byte di basso ordine sarà automaticamente immagazzinato nel bit di riporto interno ("C"). Poi verrà sommata la metà alta dei due operandi e del riporto e il risultato sarà salvato nella memoria. Il programma è il seguente:

```
LD A, (ADR1)        CARICA LA METÀ BASSA DI OP1
LD HL, ADR2         INDIRIZZA LA METÀ BASSA DI OP2
ADD A, (HL)         SOMMA OP1 E OP2 BASSI
LD (ADR3), A        IMMAGAZZINA RISULTATO, BASSO
LD A, (ADR1-1)      CARICA METÀ ALTA DI OP1
DEC HL              INDIRIZZA LA METÀ ALTA DI OP2
ADC A, (HL)         (OP1 + OP2) ALTO + RIPORTO
LD (ADR3-1), A      IMMAGAZZINA RISULTATO, ALTO
```

Le prime quattro istruzioni di questo programma sono identiche a quelle per l'addizione ad 8 bit nella parte precedente. Esse sommano le metà meno significative (bit 0-7) di OP1 e OP2. La somma, chiamata "RES" è immagazzinata nella posizione di memoria ADR3 (vedere la Figura 3-5).

Automaticamente, ogni volta che è eseguita una addizione, ogni riporto risultante (che sia "0" o "1") è salvato nel bit di riporto C del registro dei flag (registro F). Se i due numeri generano un riporto, allora il bit C sarà uguale a "1" (sarà "settato"). Se i due numeri di 8 bit non generano nessun riporto, il valore del bit di riporto sarà "0".

Le prossime quattro istruzioni del programma sono essenzialmente come quelle usate nel precedente programma dell'addizione ad 8 bit. Questa volta sommano la metà più significativa (o metà alta, cioè, i bit 8-15) di OP1 e OP2, più il riporto, ed immagazzinano il risultato nell'indirizzo ADR3-1.

Dopo l'esecuzione di questo programma di 8 istruzioni, il risultato a 16 bit è immagazzinato nelle posizioni di memoria ADR3 e ADR3-1, come specificato. Notate, comunque, che c'è una differenza tra la seconda metà di questo programma e la prima metà. L'*istruzione* "ADD" che è stata usata non è la stessa della prima metà. Nella prima metà di questo programma (la terza istruzione) noi avevamo usato l'istruzione "ADD". Questa istruzione somma i due operandi, senza curarsi del riporto. Nella seconda metà, usiamo l'istruzione "ADC", la quale somma i due operandi, più il riporto che può essere stato generato. Ciò è necessario per ottenere il risultato corretto. L'addizione inizialmente eseguita sugli operandi bassi può generare un riporto. Un tale possibile riporto deve essere tenuto in conto nella seconda metà dell'addizione.

Allora, la domanda che viene naturale è: cosa avviene se l'addizione della metà alta degli operandi porta anch'essa ad un riporto? Ci sono due possibilità: la prima è presumere che questo sia un errore. Questo programma è stato progettato per funzionare con risultati solo fino a 16 bit, ma non 17. L'altro è di includere istruzioni addizionali per verificare esplicitamente la possibilità di un riporto alla fine di questo programma. Questa è una scelta che il programmatore deve prendere, la prima di molte scelte.

Notate: qui abbiamo presupposto che la parte alta dell'operando sia immagazzinata "in cima alla" parte più bassa, cioè, all'indirizzo di memoria più basso. Questo non deve necessariamente essere. Infatti, gli indirizzi sono immagazzinati dallo Z80 nella maniera inversa: la parte bassa è conservata nella memoria per prima e la parte alta è conservata nella posizione di memoria seguente. In modo da usare una convenzione comune sia per gli indirizzi che per i dati, si raccomanda che anche i dati siano tenuti con la parte bassa sopra alla parte alta. Ciò è illustrato nella Figura 3-6.

Quando operate su operandi multibyte, è importante tenere in mente due regole essenziali:
- L'ordine in cui sono immagazzinati i dati nella memoria.
- Dove stanno puntando i puntatori dei dati: byte basso o byte alto.

Gli esercizi 3-2 e 3-3 sono progettati per chiarire questo punto.

Esercizio 3-2: *Riscrivete il programma precedente dell'addizione a 16 bit con le regole di memoria indicate nella Figura 3-6.*

Esercizio 3-3: *Ora presupponiamo che ADR1 non punti alla metà più bassa di OPR1 (come nelle Figure 3-5 o 3-6), ma punti alla parte più alta di OPR1. Ciò è illustrato nella Figura 3-7. Scrivete di nuovo il programma corrispondente.*

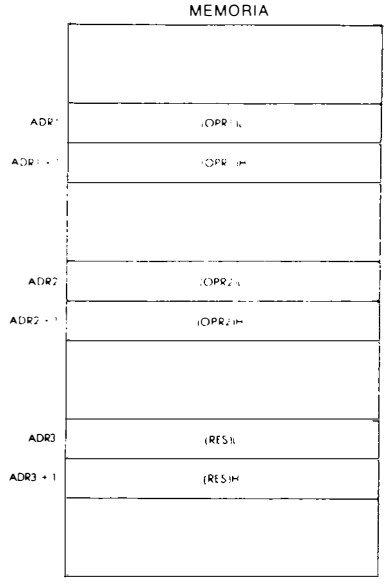


Figura 3-6: Memorizzazione degli operandi in ordine inverso

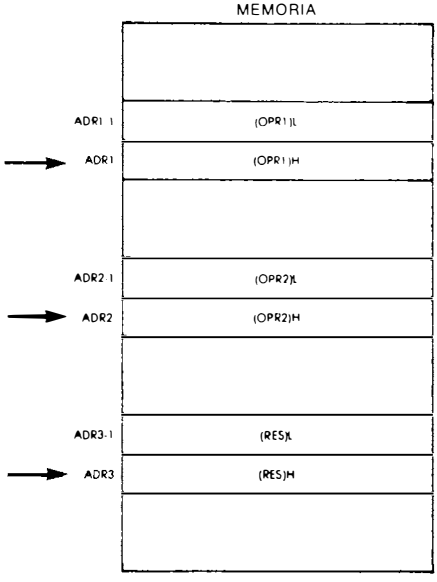


Figura 3-7: Puntatori al byte più significativo

È il programmatore, cioè, voi, che dovete decidere come immagazzinare i numeri a 16 bit (cioè, prima la parte bassa o la parte alta) e anche se i vostri riferimenti di indirizzo puntano alla metà più alta o più bassa di tali numeri. Questa è un'altra scelta che imparerete a fare quando progettate algoritmi o strutture dati.

I programmi presentati finora sono programmi tradizionali, che usano l'accumulatore. Ora presenteremo un programma alternativo per l'addizione a 16 bit che non usa l'accumulatore, ma che usa invece alcune delle speciali istruzioni a 16 bit utilizzabili sullo Z80. Sarà presupposto che gli operandi siano immagazzinati come è indicato nella Figura 3-5. Il programma è:

```
LD HL, (ADR1)      CARICA HL CON OP1
LD BC, (ADR2)      CARICA BC CON OP2
ADD HL, BC         SOMMA A 16 BIT
LD (ADR3-2), HL    IMMAGAZZINA RES IN ADR3
```

Notate quanto è più corto questo programma, paragonato alla nostra versione precedente. È più "elegante". *In una maniera limitata, lo Z80 permette che H e L siano usati come un accumulatore a 16 bit.*

Esercizio 3-4: *Usando le istruzioni a 16 bit che sono appena state introdotte, scrivete un programma per l'addizione di operandi di 32 bit, presumendo che gli operandi siano immagazzinati come mostrato nella Figura 3-8 (La risposta compare sotto).*

Risposta:

```
LD HL, (ADR1)
LD BC,(ADR2)
ADD HL,BC
LD (ADR3), HL
LD HL,(ADR1-2)
LD BC,(ADR2-2)
ADC HL, BC
LD (ADR3-2), HL
```

Ora che abbiamo imparato ad eseguire una addizione binaria, consideriamo la sottrazione.

Sottrazione dei numeri a 16 bit

Sarebbe troppo semplice fare una sottrazione ad 8 bit. Teniamola come un esercizio ed eseguiamo direttamente una sottrazione a 16 bit. Come al solito, i nostri due numeri, OP1 e OP2, sono immagazzinati agli indirizzi ADR1 e ADR2. Sarà supposto che le regole di memoria siano quelle della Figura 3-6. Per sottrarre, useremo una operazione di sottrazione (SBC) invece di una operazione di addizione (ADD).

Esercizio 3-5: *Ora scrivete un programma di sottrazione.*

Il programma compare sotto. Il percorso dei dati è mostrato nella Figura 3-9.

```
LD HL, (ADR1)      OP1 IN HL
LD DE, (ADR2)      OP2 IN DE
AND A              AZZERA RIPORTO
SBC HL, DE         OP1 – OP2
LD (ADR3), HL      RES IN ADR3
```

Il programma è essenzialmente come quello sviluppato per l'addizione a 16 bit. Il set di istruzioni dello Z80 ha due tipi di addizioni sui registri doppi: ADD e ADC, ma solo un tipo di sottrazione: SBC.

Come risultato, possono essere notate due variazioni.

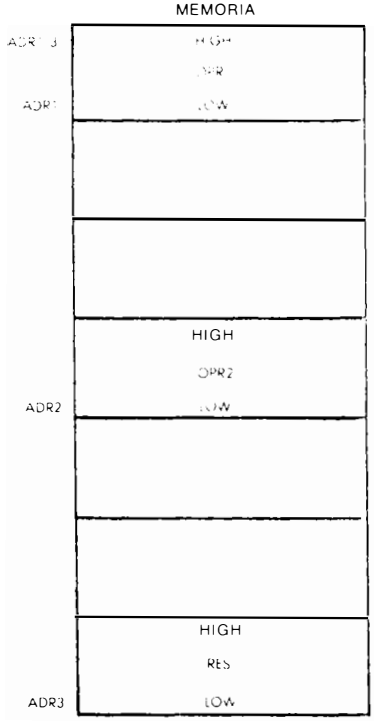


Figura 3-8: Addizione a 32 bit

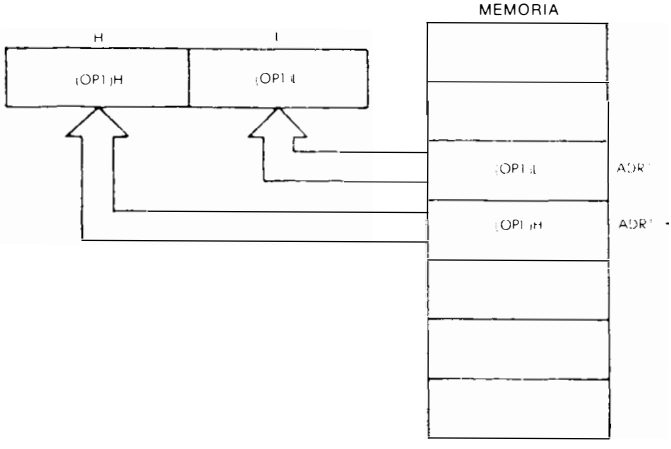


Figura 3-9: Caricamento a 16 bit — LD HL, (ADR1)

Una prima variazione è l'uso di SBC invece di ADD. L'altra variazione è l'istruzione "AND A" usata per azzerare il flag di riporto prima della sottrazione. Questa istruzione non modifica il valore di A.

Questa precauzione è necessaria perchè lo Z80 è fornito di due modi di addizione, con e senza riporto sul registro H ed L, ma con un solo modo di sottrazione, l'istruzione SBC del "sottrarre con riporto" operante sulla coppia di registri HL. Poichè l'SBC tiene conto automaticamente del valore del bit di riporto, questo deve essere portato a 0 prima di iniziare la sottrazione. Questo è il ruolo della istruzione "AND A".

Esercizio 3-6: *Riscrivete il programma della sottrazione senza usare l'istruzione specializzata a 16 bit.*

Esercizio 3-7: *Scrivete il programma della sottrazione per gli operandi ad 8 bit.*

Deve essere ricordato che nel caso dell'aritmetica in complemento a due, il valore finale del flag di riporto non ha significato. Se è avvenuta una condizione di overflow come risultato della sottrazione, allora il bit di overflow (bit V) del registro dei flag sarà portato a 1. Questa condizione può allora essere verificata. Gli esempi appena presentati sono sottrazioni o addizioni binarie semplici. Comunque, può essere necessario un altro tipo di aritmetica; è l'aritmetica BCD.

ARITMETICA BCD

Addizione BCD ad 8 Bit

Il concetto dell'aritmetica BCD è stato presentato nel Capitolo 1. Richiamiamo alla mente le sue caratteristiche. È essenzialmente usata per le applicazioni commerciali dove è imperativo trattenere ogni digit significativo di un risultato. Nella notazione BCD, un "nibble" di 4 bit viene impiegato per immagazzinare un digit decimale (da 0 a 9). Come risultato, ogni byte di 8 bit può immagazzinare due digit BCD. (Questo è chiamato BCD compattato). Ora sommiamo due byte ognuno contenente due digit BCD.

Per identificare i problemi, prima proviamo alcuni esempi numerici.

Sommiamo "01" e "02":

"01" è rappresentato da:	0000 0001
"02" è rappresentato da:	0000 0010
Il risultato è	0000 0011

Questa è la rappresentazione BCD per "03". (Se vi sentite insicuri dell'equivalente BCD, riferitevi alla tabella di conversione alla fine del libro). In questo caso tutto ha funzionato molto semplicemente. Adesso proviamo un altro esempio.

"08" è rappresentato da:	0000 1000
"03" è rappresentato da:	0000 0011

Esercizio 3-8: *Calcolate la somma dei due numeri precedenti nella rappresentazione BCD. Cosa ottenete? (segue la risposta).*

Se ottenete "0000 1011", avete calcolato la somma *binaria* di 8 e 3. Avete veramente ottenuto 11 nel sistema *binario*. Sfortunatamente, "1011" è un *codice non consentito* nel BCD. Dovreste ottenere la rappresentazione BCD di "11", cioé, 0001 0001!

Il problema deriva dal fatto che la rappresentazione BCD usa solo le prime dieci combinazioni di 4 digit per codificare i simboli decimali da 0 a 9. Le rimanenti 6 combinazioni possibili di 4 digit non sono usate, ed il valore non consentito "1011" è una di tali combinazioni. In altre

parole, ogni volta che la somma di due digit binari è più grande di 9 allora si deve aggiungere 6 al risultato per scavalcare i 6 codici non usati.

Aggiungete la rappresentazione binaria di "6" a 1011:

```
                 1011   (risultato binario non consentito)
               + 0110   (+6)
               ---------
Il risultato è:  0001 0001
```

Questo è, veramente, "11" nella numerazione BCD! Ora abbiamo il risultato corretto.

Questo esempio illustra una delle difficoltà fondamentali del modo BCD. Si devono sostituire i sei codici mancanti. Una istruzione speciale, "DAA", chiamata "aggiustamento decimale" deve essere usata per il risultato dell'addizione binaria. (Aggiungete 6 se il risultato è maggiore di 9).

Il prossimo problema è illustrato dallo stesso esempio. Nel nostro esempio, il riporto sarà generato dal digit BCD più basso (quello più a destra) in quello più a sinistra.

Questo riporto interno deve essere preso in considerazione e sommato al secondo digit BCD. L'istruzione di addizione si prende cura di ciò automaticamente. Comunque, è spesso conveniente rivelare questo riporto interno dal bit 3 a bit 4 (il mezzo riporto). Il flag H è fornito per questo scopo.

Come esempio, ecco un programma per sommare i numeri BCD "11" e "22":

```
LD    A, 11H      CARICA IL "LITERAL" BCD "11"
ADD   A, 22H      SOMMA IL "LITERAL" BCD "22"
DAA               AGGIUSTAMENTO DECIMALE DEL RISULTATO
LD    (ADR), A    IMMAGAZZINA RISULTATO
```

In questo programma, stiamo usando un nuovo simbolo "H". Il segno "H" all'interno del campo degli operandi dell'istruzione specifica che i dati che seguono sono espressi nella numerazione esadecimale. Gli esadecimali e le rappresentazioni BCD per i digit da "0" a "9" sono identici. Noi qui desideriamo sommare i "literal" (letterali) (o costanti) "11" e "22". Il risultato è immagazzinato all'indirizzo ADR. Quando l'operando è specificato come parte dell'istruzione, come lo è per l'esempio sopra, questo è chiamato *indirizzamento immediato.* (I vari modi di indirizzamento saranno discussi dettagliatamente nel capitolo 5). Immagazzinare il risultato ad un indirizzo specificato, come LD (ADR), A è chiamato *indirizzamento assoluto* quando ADR rappresenta un indirizzo a 16 bit.

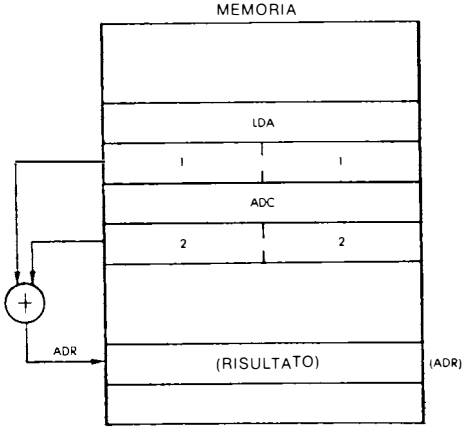


Figura 3-10: Immagazzinamento di digit BCD

Questo programma è analogo all'addizione binaria ad 8 bit, ma usa una nuova istruzione: "DAA". Illustriamo il suo ruolo in un esempio. Per primo sommiamo "11" e "22" in BCD:

```
  00010001   (11)
+ 00100010   (22)
= 00110011   (33)
   3    3
```

Usando le regole dell'addizione binaria il risultato è corretto.
Sommiamo ora "22" e "39", usando le regole dell'addizione *binaria:*

```
  00100010   (22)
+ 00111001   (39)
= 01011011
   5    ?
```

"1011" è un codice BCD non consentito. E questo perchè il BCD usa solo i primi 10 codici binari, e "scavalca" i 6 successivi. Noi dobbiamo fare lo stesso, cioè, sommare 6 al risultato:

```
  01011011   (risultato binario)
+     0110   ( 6)
  01100001   (61)
   6    1
```

Questo è il risultato BCD corretto.

Esercizio 3-9: *Potremmo spostare l'istruzione DAA nel programma dopo l'istruzione LD (ADR),A?*

Sottrazione BCD

In apparenza, la sottrazione BCD è complessa. Per eseguire una sottrazione BCD, si deve sommare il *complemento a dieci* del numero, proprio come si somma il complemento a due di un numero per eseguire una sottrazione binaria. Il complemento a dieci è ottenuto calcolando il complemento a 9, e poi sommando "1". Questo richiede tipicamente da tre a quattro operazioni su un microprocessore standard.

Comunque, lo Z80 è fornito di una potente istruzione DAA che semplifica il programma.

L'istruzione DAA regola automaticamente il valore del risultato nell'accumulatore, a seconda del valore dei flag C e H prima dell'istruzione DAA, al valore corretto. (Vedere il prossimo capitolo per altri dettagli su DAA).

Addizione BCD a 16 bit

L'addizione a 16 bit è eseguita tanto semplicemente quanto nel caso binario. Il programma per una tale addizione è il seguente:

```
LD    A,(ADR1)        CARICA (OP1) BASSO IN A
LD    HL,ADR2         CARICA ADR2 IN HL
ADD   A,(HL)          (OP1 + OP2) BASSO
DAA                   AGGIUSTAMENTO DECIMALE
LD    (ADR3),A        IMMAGAZZINA (RISULTATO) BASSO
LD    A,(ADR1 + 1)    CARICA (OP1) ALTO IN A
INC   HL              PUNTA A ADR2 + 1
ADC   A,(HL)          (OP1 + OP2) ALTO + RIPORTO
DAA                   AGGIUSTAMENTO DECIMALE
LD    (ADR3 + 1),A    IMMAGAZZINA (RISULTATO) ALTO
```

Sottrazione BCD compattata

Le addizioni e sottrazioni BCD elementari sono state descritte. Comunque, nella pratica, i numeri BCD includono qualunque numero di byte. Come esempio semplificato di una sottrazione BCD compattata, noi presumeremo che i due numeri N1 e N2 includano lo stesso numero di byte BCD. Il numero di byte è chiamato COUNT.

L'assegnazione dei registri e della memoria è mostrata nella Figura 3-11. Il programma compare sotto:

```
BCDPAK   LD     B,COUNT
         LD     DE,N2
         LD     HL,N1
         AND    A                    AZZERA RIPORTO
MINUS    LD     A,(DE)               N2 BYTE
         SBC    (A,(HL)              N2-1
         DAA
         LD     (HL),A               IMMAGAZZINA RISULTATO
         INC    DE
         INC    HL
         DJNZ   MINUS                DEC B, LOOP FINO B-0
```

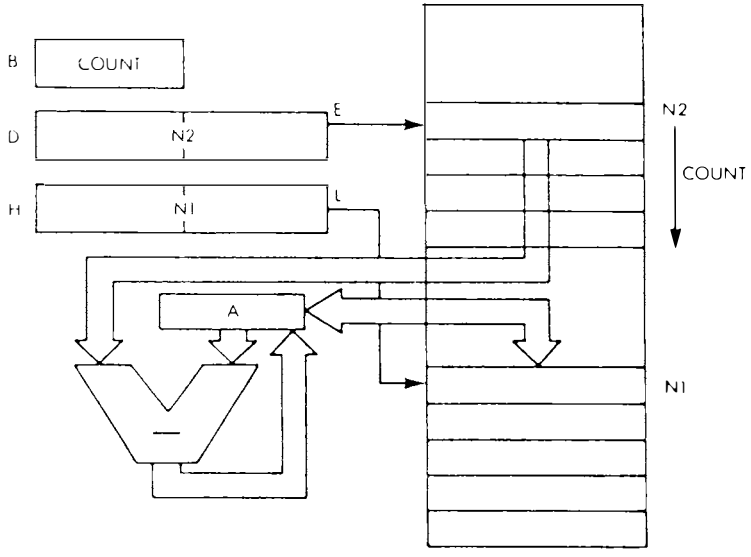


Figura 3-11: Sottrazione BCD compattata: N1 · N2 – N1

N1 ed N2 rappresentano gli indirizzi dove sono immagazzinati i numeri BCD.

Questi indirizzi saranno caricati nelle coppie di registri DE e HL:

```
BCDPAK   LD     B,COUNT
         LD     DE,N2
         LD     HL,N1
```

Poi, in previsione della prima sottrazione, il bit di riporto deve essere azzerato. È stato posto in rilievo che il bit di riporto può essere azzerato in numerosi modi equivalenti. Qui, per esempio, usiamo:

```
        AND     A
```

Il primo byte di N2 è caricato nell'accumulatore, poi il primo byte di N1 è sottratto da esso. È allora usata l'istruzione DAA, per ottenere il valore BCD corretto:

```
MINUS   LD      A, (DE)
        SBC     A, (HL)
        DAA
```

Il risultato è poi immagazzinato in N1:

```
        LD      (HL), A
```

Alla fine, sono incrementati i puntatori al byte corrente:

```
        INC     DE
        INC     HL
```

Il contatore è diminuito ed il loop di sottrazione è eseguito fino a quando raggiunge il valore "0":

```
        DJNZ    MINUS
```

L'istruzione DJNZ è una istruzione dello Z80 speciale che diminuisce il registro B e salta se non è zero, in una istruzione singola.

Esercizio 3-10: *Paragonate il programma precedente a quello per l'addizione binaria a 16 bit. Qual è la differenza?*

Esercizio 3-11: *Potete scambiare il ruolo di DE e HL? (Suggerimento: Attenti con SBC).*

Esercizio 3-12: *Scrivete un programma di sottrazione per BCD a 16 bit.*

I Flag BCD

Nel sistema BCD, il flag di riporto durante un'addizione indica il fatto che il risultato è più grande di 99. Questo non è come la situazione del complemento a due dal momento in cui i digit BCD sono rappresentati in vero binario. Viceversa, la presenza del flag di riporto durante una sottrazione indica un prestito.

TIPI D'ISTRUZIONE

Ora abbiamo usato due tipi di istruzioni per microprocessori.
Abbiamo usato LD, la quale carica l'accumulatore dall'indirizzo di memoria, o immagazzina il suo contenuto all'indirizzo specificato. Questa è una istruzione di *trasferimento dati*.

Poi abbiamo usato istruzioni *aritmetiche*, come ADD, SUB, ADC e SBC. Eseguono operazioni di addizione e sottrazione. In questo capitolo saranno presto introdotte altre istruzioni della ALU.

Sono disponibili altri tipi di istruzioni all'interno del microprocessore che non sono ancora state usate. Sono in particolare le istruzioni "jump", che modificheranno l'ordine in cui è eseguito il programma. Questo nuovo tipo di istruzione sarà introdotta nel nostro prossimo esempio. Notate che le istruzioni jump sono spesso chiamate "branch" (diramazioni), per situazioni condizionali, cioè, casi dove c'è una scelta logica nel programma. Il "branch" deriva il suo nome dalla analogia con l'albero, ed implica una biforcazione nella rappresentazione del programma.

MOLTIPLICAZIONE

Esaminiamo adesso un problema aritmetico più complesso: la moltiplicazione dei numeri binari. Per introdurre l'algoritmo per una moltiplicazione binaria, iniziamo esaminando una moltiplicazione decimale comune: Moltiplicheremo 12 per 23.

```
   12   (moltiplicando)
 × 23   (moltiplicatore)
   36   (prodotto parziale)
 +24
  276   (risultato finale)
```

La moltiplicazione viene eseguita moltiplicando il digit più a destra del moltiplicatore per il moltiplicando, cioè, "3" × "12". Il prodotto parziale è "36". Poi si moltiplica il digit seguente del moltiplicatore, cioè, "2", per "12". "24" è poi aggiunto al prodotto parziale.

Ma c'è un'altra operazione: 24 è *sfalsata alla sinistra* di una posizione. Diremo che 24 è *spostato a sinistra* di una posizione. Allo stesso modo, potremmo dire che il prodotto parziale (36) è stato *spostato di una posizione alla destra* prima di addizionare.

I due numeri, correttamente spostati, sono poi addizionati e la somma è 276. Ciò è semplice. La moltiplicazione binaria è eseguita esattamente nello stesso modo.

Guardiamo un esempio. Moltiplicheremo 5 × 3:

```
(5)        101   (MPD)
(3)       ×011   (MPR)
           101   (PP)
          101
         000
(15)     01111   (RES)
```

Per eseguire la moltiplicazione, operiamo esattamente come abbiamo fatto qui sopra. La rappresentazione formale di questo algoritmo compare nella Figura 3-12. È un diagramma di flusso (flowchart) per l'algoritmo, il nostro primo diagramma di flusso. Esaminiamolo più attentamente.

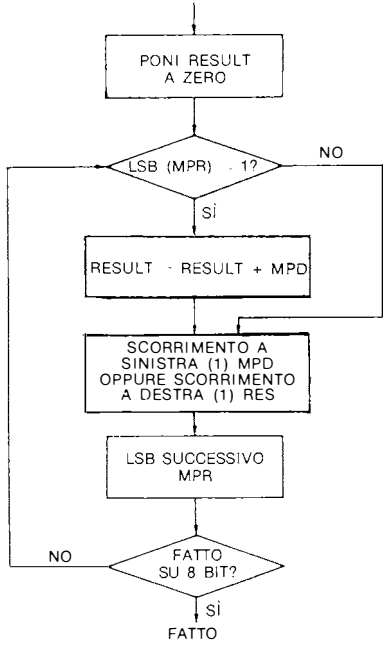


Figura 3-12: Algoritmo di base della moltiplicazione: Diagramma di flusso

Questo flowchart è una rappresentazione simbolica dell'algoritmo che abbiamo appena presentato. Ogni rettangolo rappresenta un ordine che deve essere eseguito. Sarà tradotto in uno o più istruzioni di programma.

Ogni simbolo a forma di rombo rappresenta un test che deve essere eseguito. Questo sarà un *punto di biforcazione* nel programma. Se il test riesce, noi salteremo ad una posizione specificata. Se il test non riesce, salteremo ad un'altra posizione.

Il concetto di biforcazione sarà spiegato più tardi, nello stesso programma. Il lettore ora dovrebbe esaminare questo flowchart e assicurarsi che rappresenti veramente in modo esatto l'algoritmo che è stato presentato. Notate che c'è una freccia che esce dall'ultimo rombo nella parte bassa del flowchart e torna indietro al primo rombo in alto. Questo è perché la stessa porzione del flowchart sarà eseguita otto volte, una volta per ognl bit del moltiplicatore. Una tale situazione, dove l'esecuzione inizierà di nuovo allo stesso punto, è chiamata "program loop" (ciclo di programma) per ovvie ragioni.

Esercizio 3-13: *Moltiplicate "4" per "7" in binario, usando il flowchart, e verificate che si ottenga "28". Se non l'ottenete, provate di nuovo. È soltanto se ottenete il risultato corretto che voi siete pronti a tradurre questo flowchart in un programma.*

Moltiplicazione 8 x 8

Traduciamo adesso questo flowchart in un programma per lo Z80. Il programma completo compare nella Figura 3-13. Lo studieremo dettagliatamente. Come vi ricorderete dal Capitolo 1, la programmazione qui consiste nel tradurre il flowchart della Figura 3-12 nel programma della Figura 3-13. Ognuno dei blocchi nel flowchart sarà tradotto in una o più istruzioni. È presupposto che MPR e MPD abbiano già un valore.

```
MPY88   LD    BC (MPRAD)    CARICA IL MOLTIPLICATORE IN C
        LD    B,8           B È IL CONTATORE DI BIT
        LD    DE, (MPDAD)   CARICA IL MOLTIPLICANDO IN E
        LD    D,0           AZZERA D
        LD    HL,0          AZZERA IL RISULTATO
MULT    SRL   C             SPOSTA IL BIT DEL MOLTIPLICATORE
                            NEL RIPORTO
        JR    NC,NOADD      TEST DEL RIPORTO
        ADD   HL,DE         AGGIUNGI MPD AL RISULTATO
NOADD   SLA   E             SPOSTA MPD A SINISTRA
        RL    D             SALVA BIT IN D
        DEC   B             DIMINUISCI CONTATORE DI SPOSTAMENTO
        JP    NZ,MULT       RIPETI SE IL CONTATORE≠ 0
        LD    (RESAD),HL    IMMAGAZZINA RISULTATO
```

Figura 3-13: Programma di moltiplicazione 8 × 8

Il primo blocco del flowchart è un blocco *di inizializzazione*. È necessario per azzerare un numero di registri o di posizioni di memoria, in quanto questo programma richiederà il loro uso. I registri che saranno usati dal programma di moltiplicazione compaiono nella Figura 3-14.

Tre coppie di registri dello Z80 sono usate per il programma di moltiplicazione. Il moltiplicatore di 8 bit si assume che risieda all'indirizzo di memoria MPRAD. Il moltiplicando MPD si assume che risieda all'indirizzo di memoria MPDAD. Il moltiplicatore e il moltiplicando saranno rispettivamente caricati nei registri C ed E (vedere la Figura 3-14). Il registro B sarà usato come contatore.

I registri D e E conterranno il moltiplicando quando è spostato a sinistra un bit alla volta.

Notate che, anche se solo C e E hanno bisogno di essere caricati inizialmente, deve essere usato un caricamento a 16 bit, in modo che anche B e D saranno caricati dalla memoria, e dovranno essere rispettivamente portati a "8" e a "0".

Alla fine, i risultati di una moltiplicazione da 8 bit per 8 bit può richiedere fino a 16 bit. Questo perchè $2^8 \times 2^8 = 2^{16}$. Devono perciò essere riservati per il risultato due registri.

Sono i registri H ed L, come indicato nella Figura 3-14.

La prima fase è di caricare i registri B, C ed E con il contenuto appropriato, ed inizializzare il risultato (il prodotto parziale) al valore "0" come specificato dal flowchart della Figura 3-12.

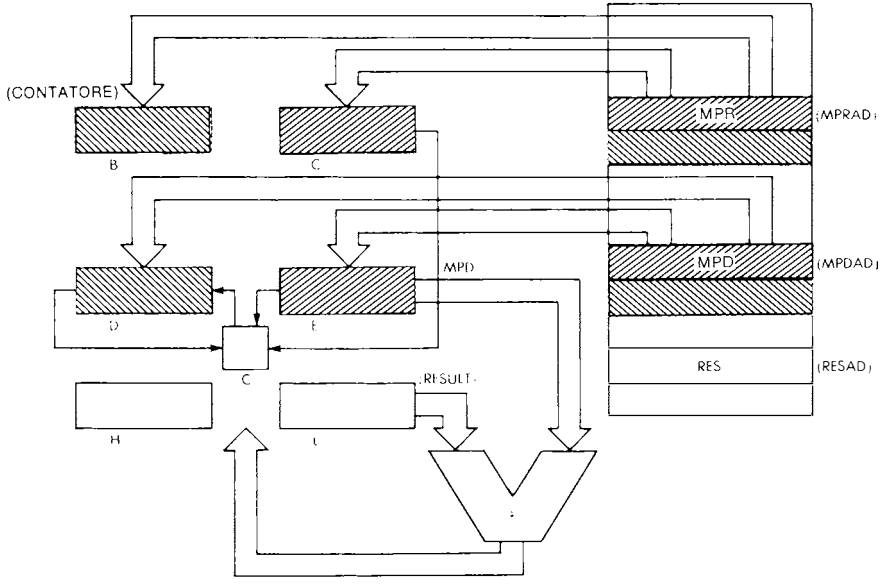


Figura 3-14: Moltiplicazione 8 × 8 — I registri

Ciò è effettuato dalle seguenti istruzioni:

```
MPY88   LD   BC, (MPRAD)
        LD   B, 8
        LD   DE, (MPDAD)
        LD   D, 0
        LD   HL, 0
```

Le prime tre istruzioni rispettivamente caricano MPR nella coppia di registri BC, il valore "8" nel registro B, ed MPD nella coppia di registri DE. Poichè MPR e MPD sono parole di 8 bit, essi sono infatti, caricati rispettivamente nei registri C e E, mentre le parole di memoria dopo MPR e MPD sono caricati in B e D. Ciò è mostrato nella Figura 3-15, e 3-16. L'istruzione successiva azzererà il contenuto in D.

Nel programma di moltiplicazione, il moltiplicando sarà spostato a sinistra prima di essere sommato al risultato (ricordatevi che, opzionalmente, è possibile spostare invece il risultato a destra, come indicato nel quarto blocco del flowchart della Figura 3-12). Il moltiplicando MPD sarà spostato nel registro D ad ogni fase. Questo registro D deve perciò essere inizializzato al valore "0". Ciò è eseguito dalla quarta istruzione. Alla fine, la quinta istruzione pone il contenuto dei registri H ed L a 0 con una istruzione singola.

Riferendoci al flowchart della Figura 3-12, la prossima fase è quella di verificare il bit meno significativo (il bit più a destra) del moltiplicatore MPR. Se questo bit è un "1", allora il valore di MPD deve essere sommato al risultato parziale, altrimenti non sarà sommato. Ciò è eseguito dalle prossime tre istruzioni:

```
MULT    SRL     C
        JR      NC,NOADD
        ADD     HL,DE
```

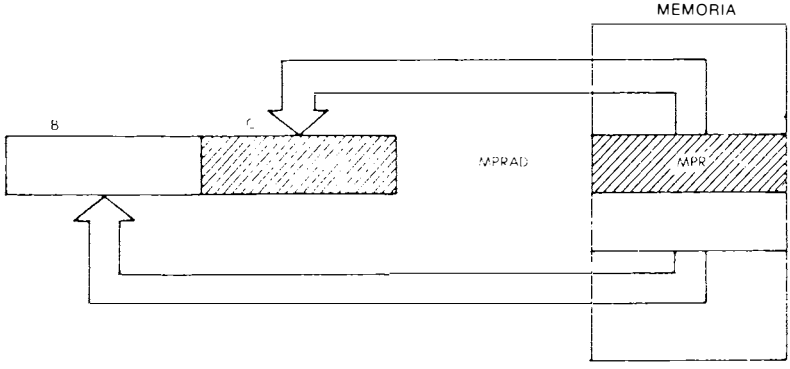


Figura 3-15: LD BC, (MPRAD)

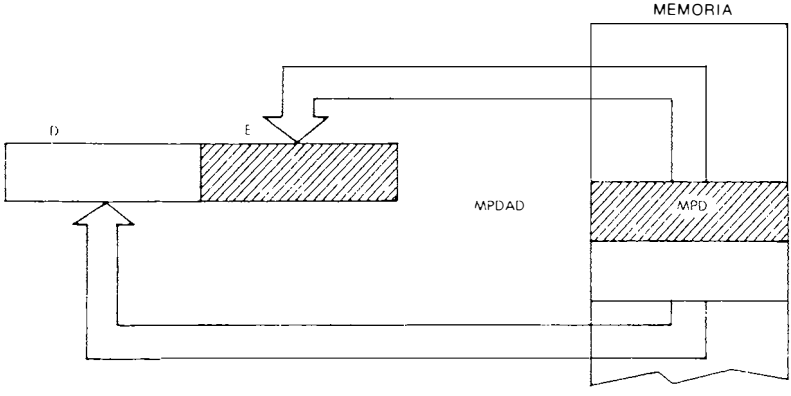


Figura 3-16: LD DE, (MPDAD)

Il primo problema che dobbiamo risolvere è come provare il bit meno significativo del moltiplicatore, contenuto nel registro C. Qui potremmo usare l'istruzione BIT dello Z80, la quale permette di provare qualsiasi BIT in qualsiasi registro. Comunque, in questo caso, ci piacerebbe costruire un programma quanto più semplice possibile, usando un loop. Se qui stessimo u-

sando l'istruzione BIT, per primo proveremmo il bit 0, e dopo proveremmo il bit 1, e così via fino a che raggiungiamo il bit 7. Questo richiederebbe una diversa istruzione ogni volta, e non potrebbe essere usato un semplice loop. Per accorciare la lunghezza del programma, dobbiamo usare una istruzione differente. Qui, stiamo usando una istruzione "shift".

Notate: c'è un modo di usare l'istruzione BIT e un loop, ma questo richiederebbe che il programma modifichi se stesso, una pratica che eviteremo.

SRL è un nuovo tipo di operazione all'interno dell'unità aritmetico-logica. Sta per "shift right logical" (spostamento logico a destra). Uno *spostamento logico* è caratterizzato dal fatto che uno "0" entra nella posizione di bit 7. Questo a differenza di uno *spostamento aritmetico*, dove il bit che entra nella posizione 7 è identico al precedente valore del bit 7. I differenti tipi di operazioni di spostamento saranno descritti nel prossimo capitolo. L'effetto dell'istruzione SRLC è illustrato nella Figura 3-14 da una freccia che esce dal registro C e entra nel quadratino usato per designare il bit di riporto (chiamato anche "C"). A questo punto, il bit più a destra dell'MPR sarà nel bit di riporto C, dove può essere provato.

La prossima istruzione, "JR NC, NOADD" è un'operazione di *jump* (salto). Significa "jump on no carry" (saltare se non c'è riporto) (NC) all'indirizzo (la label) NOADD. Se il contenuto del bit di riporto è "0" (nessun riporto), allora il programma salterà all'indirizzo NOADD. Se il contenuto di C è "1" (il bit del riporto è ad "1"), allora non accadrà nessuna biforcazione e sarà eseguita la successiva istruzione sequenziale, cioè, sarà eseguita l'istruzione "ADD HL,DE". Questa istruzione specifica che il contenuto di D ed E sia sommato ad H ed L, con il risultato in H ed L. Poiché E contiene il moltiplicando MPD (vedere la Figura 3-14), si somma il moltiplicando al risultato parziale.

A questo punto, senza curarsi del fatto se MPD è stato sommato al risultato o no, il moltiplicando deve essere spostato a sinistra (questo è il quarto blocco nel flowchart di Figura 3-12). Ciò è eseguito da:

```
NOADD   SLA   E
```

SLA sta per spostamento aritmetico a sinistra. È stato appena spiegato sopra che ci sono due tipi di operazioni di spostamento, uno spostamento logico (logical shift) e uno spostamento aritmetico (arithmetic shift). Questo è quello aritmetico. Nel caso di uno spostamento a sinistra, un SLA specifica che il bit che entra nella parte destra del registro (il bit meno significativo) sia uno "0" (proprio come prima nel caso di un SRL).

Come esempio, presumiamo che il contenuto iniziale del registro E fosse 00001001. Dopo l'istruzione SLA, il contenuto di E sarà 00010010. E il contenuto del bit del riporto sarà 0.

Comunque, riguardando la Figura 3-14, noi vogliamo veramente spostare il bit più significativo (chiamato MSB) di E direttamente in D (ciò è illustrato nell'illustrazione da una freccia che va da E in D). Comunque, non c'è nessuna istruzione che sposterà un registro doppio come D ed E in una sola operazione. Una volta che il contenuto di E è stato spostato, il bit più a sinistra è "caduto" nel bit del riporto! Dobbiamo raccogliere questo bit dal bit del riporto e spostarlo nel registro D. Questo è eseguito dalla prossima istruzione:

```
RL   D
```

RL è ancora un altro tipo di operazione di spostamento. Sta per "rotate left" (rotazione a sinistra). In una operazione di *"rotate"* in maniera opposta ad una operazione di *shift*, questo bit che viene nel registro è il contenuto del bit del riporto C (vedere la Figura 3-17). Ciò è esattamente quello che vogliamo. Il contenuto del bit del riporto C è caricato nella parte più a destra di D, ed abbiamo effettivamente trasferito il bit più a sinistra di E.

Questa sequenza di due istruzioni è illustrata nella Figura 3-18. Può essere visto che il bit segnato con una X nella posizione più significativa di E sarà per prima cosa trasferito nel bit di riporto, e poi nella posizione meno significativa di D. Effettivamente, sarà stato spostato da E in D.

A questo punto, riferendoci al flowchart della figura 3-12, dobbiamo puntare al prossimo bit

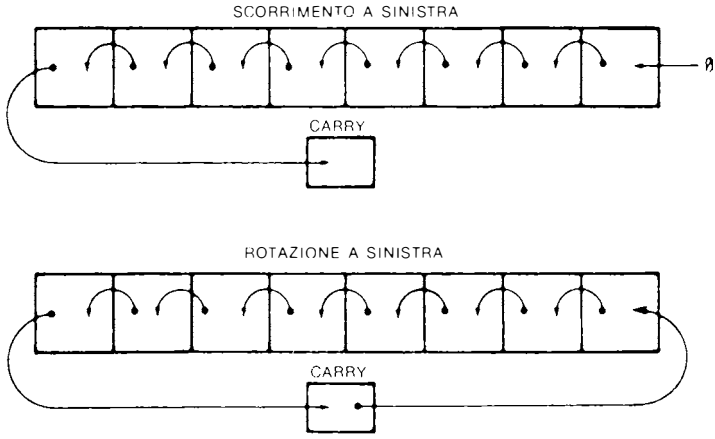


Figura 3-17: Scorrimento e Rotazione

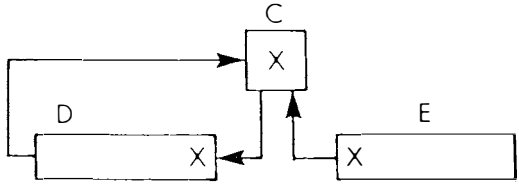


Figura 3-18: Spostamento da E in D

di MPR e fare un check sull'ottavo bit. Ciò è eseguito diminuendo il contatore del byte, contenuto nel registro B (vedere Figura 3-14). Il registro è diminuito tramite:

```
DEC  B
```

Questa è un'istruzione di "*decremento*" che ha l'effetto ovvio.

Alla fine, dobbiamo controllare se il contatore è diminuito al valore zero. Ciò è eseguito controllando il valore del bit Z. Il lettore si ricorderà che il flag Z (zero) indica se la precedente operazione aritmetica (come l'operazione DEC) ha prodotto un risultato zero. Comunque, notate che DECHL, DECBC, DECDE, DECIX, DECSP non interessano il flag Z.

Se il contatore non è "0", l'operazione non è finita, e noi dobbiamo eseguire questo loop di nuovo. Ciò è eseguito dalla prossima istruzione:

```
JP  NZ  MULT
```

Questa è una istruzione jump che specifica che ogni volta che il bit Z non è ad "1" (NZ sta per non zero), avviene un salto alla posizione MULT. Questo è il *loop*, che sarà eseguito ripetutamente fino a che B diminuisce al valore 0. Ogni volta che B diminuisce al valore 0, il bit Z

sarà ad "1" e l'istruzione JP NZ non farà saltare. E quindi si passerà ad eseguire la prossima istruzione sequenziale, e cioè:

```
LD   (RESAD),  HL
```

Questa istruzione salverà soltanto il contenuto di H ed L, cioè, il risultato della moltiplicazione, all'indirizzo RESAD, l'indirizzo specificato per il risultato. Notate che questa istruzione trasferirà il contenuto di entrambi i registri H ed L in due posizioni di memoria consecutive, che corrispondono agli indirizzi RESAD e RESAD + 1. Essa salva 16 bit alla volta.

Esercizio 3-14: *Potreste scrivere lo stesso programma di moltiplicazione usando l'istruzione BIT (descritta nel prossimo capitolo) invece dell'istruzione SRL C? Quale sarebbe lo svantaggio?*

Se è possibile, miglioriamo il programma:

Esercizio 3-15: *JR può sostituire JP alla fine del programma? Se è così qual è il vantaggio?*

Esercizio 3-16: *Potete usare DJNZ per accorciare la fine del programma?*

Esercizio 3-17: *Esaminate le due istruzioni: LD D, 0 e LD HL, 0 all'inizio del programma. Potete sostituire:*

```
XOR   A
LD    D, A
LD    H, A
LD    L, A
```

Se sì, qual è l'impatto sulla dimensione (numero di byte) e sulla velocità?

Notate che, nella maggior parte dei casi, il programma che abbiamo appena sviluppato sarà una subroutine (sottoprogramma) e l'istruzione finale della subroutine sarà RET (ritorno). Il meccanismo delle subroutine sarà spiegato più avanti in questo capitolo.

Importante Self-Test

Questo è il primo programma significativo che abbiamo incontrato finora. Include molti differenti tipi di istruzioni, precisamente le istruzioni di trasferimento (LD), le operazioni aritmetiche (ADD), le operazioni logiche (SRL, SLA, RL), e le operazioni jump (JR, JP). Realizza anche un "program loop", in cui le sette istruzioni più basse, iniziando all'indirizzo MULT, sono eseguite ripetutamente. Per capire la programmazione, è essenziale capire il modo d'agire di tale programma nei più completi dettagli. Il programma è molto più lungo dei precedenti programmi aritmetici semplici che abbiamo sviluppato finora, e dovrebbe essere studiato dettagliatamente. Ora sarà proposto un importante esercizio. Il lettore è fortemente raccomandato di fare questo esercizio completamente e correttamente prima di andare avanti. Questo sarà la sola vera prova che siano stati capiti i concetti presentati finora. Se è ottenuto un risultato corretto, significherà che avrete veramente capito il meccanismo col quale le istruzioni manipolano le informazioni nel microprocessore, le trasferiscono tra la memoria e i registri e le trattano. Se non ottenete il risultato corretto, o se non fate questo esercizio, è probabile che voi troviate delle difficoltà più avanti nello scrivere dei programmi da soli. Imparare a programmare richiede pratica personale. Ora fate una pausa nella lettura, prendete un pezzo di carta, oppure usate l'illustrazione della Figura 3-19, e fate il prossimo esercizio:

LABEL	ISTRUZIONE	B	C	C (RIPORTO)	D	E	H	L

Figura 3-19: Tabella per esercizio sulle moltiplicazioni

Esercizio 3-18: *Ogni volta che viene scritto un programma, dovrebbe essere verificato attentamente, per assicurarsi che il suo risultato sarà corretto. Stiamo proprio per fare questo: lo scopo di questo esercizio è di riempire accuratamente e completamente la tabella della Figura 3-19.*

Potete scrivere direttamente alla Figura 3-19, oppure farne una copia. Dobbiamo determinare il contenuto di ogni registro relativo nello Z80 dopo l'esecuzione di ogni istruzione nel programma, dall'inizio alla fine. Tutti i registri usati dal programma della Figura 3-13 sono mostrati nella Figura 3-19. Dalla sinistra alla destra, sono i registri B e C, il riporto C, i registri D ed E ed, alla fine, i registri H ed L. Nella parte sinistra di questa illustrazione, riportate la label, se è applicabile, e poi le istruzioni da eseguire.

Alla destra dell'istruzione, inserite il contenuto di ogni registro dopo l'esecuzione dell'illustrazione. Ogni volta che il contenuto di un registro non è conosciuto (indefinito), potreste usare una lineetta per rappresentare il suo contenuto. Cominciamo riempiendo questa tabella insieme. Dovrete poi riempirla da soli fino alla fine. La prima linea è la seguente:

LABEL	ISTRUZIONE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--

Figura 3-20: Moltiplicazione: Dopo una istruzione

Qui, presupporremo di stare moltiplicando "3" (MPR) per "5" (MPD).

La prima istruzione da eseguire è "LD BC, (MPRAD)" Il contenuto della posizione di memoria MPRAD è caricato nei registri B e C. È presupposto che MPR sia uguale a 3, cioè, "00000011". Dopo l'esecuzione di questa istruzione, il contenuto del registro C è stato posto a "3".

Notate che questa istruzione avrà lo stesso risultato anche caricando il registro B con qualunque cosa che seguiva MPR nella memoria. Comunque, la prossima istruzione nel programma si occuperà di questo caricando il registro B con "8", come mostrato nella Figura 3-21. Notate che, a questo punto, il contenuto di D ed E, H ed L è ancora indefinito, e ciò è indicato da lineette. L'istruzione LD non condiziona il bit del riporto, in modo che il contenuto del bit del riporto C è indefinito. Anche questo è indicato da una lineetta.

LABEL	ISTRUZIONE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--

Figura 3-21: Moltiplicazione: Dopo due istruzioni

La situazione dopo l'esecuzione delle prime cinque istruzioni del programma (appena prima di MULT) è mostrata nella Figura 3-22.

LABEL	ISTRUZIONE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Figura 3-22: Moltiplicazione: Dopo cinque istruzioni

L'istruzione SRL eseguirà uno spostamento logico a destra, ed il bit più a destra di MPR cadrà nel bit del riporto. Potete vedere nella Figura 3-23 che il contenuto di MPR dopo lo spostamento è "00000001". Il bit del riporto C è adesso posto ad "1". Gli altri registri non sono cambiati durante questa operazione. Si raccomanda di continuare a riempire il grafico da soli.

Una seconda iterazione è mostrata alla fine di questo capitolo nella Figura 3-41.

LABEL	ISTRUZIONE	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MP488	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

Figura 3-23: Un intero attraversamento del Loop

La Figura 3-39, alla fine di questo capitolo, riporta un elenco completo dei contenuti di tutti i registri dello Z80 ed i flag per la moltiplicazione completa. Nella Figura 3-40 è mostrata una elencazione esadecimale o decimale.

Alternative di programmazione

Il programma che abbiamo appena sviluppato avrebbe potuto essere scritto in molti altri modi. Come regola generale, ogni programmatore può usualmente trovare modi per modificare, e spesso migliorare, un programma. Per esempio, abbiamo spostato il moltiplicando a sinistra prima di sommare. Sarebbe stato matematicamente equivalente spostare il risultato di una posizione alla destra prima di aggiungerlo al moltiplicando. In effetti, questo è un esercizio interessante!

Esercizio 3-19: *Scrivete un programma di moltiplicazione 8 × 8 usando lo stesso algoritmo, ma spostando il risultato di una posizione alla destra invece di spostare il moltiplicando di una posizione alla sinistra. Confrontatelo con il programma precedente, e determinate se questo differente approccio è più veloce o più lento del precedente. Le velocità delle istruzioni dello Z80 sono date nel prossimo capitolo e anche nell'appendice.*

Programma di moltiplicazione migliorato

Il programma che abbiamo appena sviluppato è una traduzione diretta dell'algoritmo da codificare. Comunque, la *programmazione efficace richiede un'accurata attenzione* del dettaglio, e la lunghezza del programma può essere spesso ridotta oppure la sua velocità di esecuzione può essere migliorata. Adesso studieremo alternative pensate per migliorare questo programma fondamentale.

Passo 1

Un primo miglioramento possibile sta nella migliore utilizzazione del set di istruzioni dello

Z80. La penultima istruzione così come quella precedente, può essere sostituita da una istruzione singola:

```
DJNZ   LOOP
```

Questo è uno speciale "jump automatizzato" dello Z80 il quale diminuisce il registro B e salta ad una posizione specificata se B non è "0". Per essere assolutamente corretti, l'istruzione non è completamente identica alla coppia precedente:

```
DEC   B
JP NZ,   MULT
```

perché specifica uno *spostamento massimo*, e si può saltare solo entro la gamma di 256 a - 256 istruzioni.

Comunque, qui dobbiamo saltare ad una posizione che è lontana soltanto alcuni byte, e questo miglioramento è legittimo. Il programma che ne risulta è mostrato sotto nella Figura 3-24:

```
MP488B   LD      DE, (MPDAD)
         LD      BC, (MPRAD)
         LD      B, 8          CONTATORE DI BIT
         LD      HL, 0
MULT     SRL     C
         JR      NC, NOADD
         ADD     HL, DE
NOADD    SLA     E
         RL      D
         DJNZ    MULT
         LD      (RESAD), HL
         RET
```

Figura 3-24: Moltiplicazione migliorata, Passo 1

Passo 2

Per migliorare ulteriormente questo programma di moltiplicazione, osserveremo che sono state usate tre differenti operazioni di spostamento nel programma iniziale della Figura 3-13. Il moltiplicatore è spostato a destra, poi il moltiplicando MPD è spostato a sinistra, in due operazioni, spostando per primo il registro E a sinistra, e poi ruotando il registro D a sinistra. Questo è spreco di tempo. Un "trucco" di programmazione standard usato nel caso della moltiplicazione è basato sulla seguente osservazione: ogni volta che il moltiplicatore è spostato di una posizione, diventa disponibile un'altra posizione di bit nel registro del moltiplicatore. Per esempio, presumiamo che il moltiplicatore si sposti a destra (nell'esempio precedente), diventa disponibile a sinistra una posizione di bit. Simultaneamente si può osservare che il primo prodotto parziale (o "risultato") userà al massimo 9 bit.

Se un registro singolo è stato assegnato al risultato all'inizio del programma, noi potremmo allora usare la posizione del bit che è lasciata vacante dal moltiplicatore per immagazzinare il quarto bit del risultato.

Dopo il prossimo spostamento dell'MPR la dimensione del prodotto parziale sarà aumentata di nuovo soltanto di un bit. In altre parole, un registro singolo può essere riservato inizialmente per il prodotto parziale, e le posizioni dei bit che sono liberate dal moltiplicatore possono allora essere usate quando l'MPR è spostato. Per migliorare il programma assegneremo perciò MPR e RES ad una coppia di registri.

Idealmente, dovrebbero essere spostati insieme in una operazione singola. Sfortunatamente lo Z80 sposta solo registri ad 8 bit in una volta. Come la maggior parte dei microprocessori ad 8 bit, non ha nessuna istruzione che permetta lo spostamento di 16 bit alla volta.

Comunque, può essere usato un altro trucco. Lo Z80 (come l'8080) è fornito di speciali istruzioni di addizione a 16 bit che abbiamo già usato.

Purchè il moltiplicatore ed il risultato siano immagazzinati nella coppia di registri H ed L, noi possiamo usare l'istruzione:

```
ADD   HL,   HL
```

la quale aggiunge il contenuto di H ed L a sè stesso. Aggiungere un numero uguale a sè stesso equivale a raddoppiarlo. Raddoppiare un numero nel sistema binario è equivalente ad uno spostamento a sinistra. Abbiamo appena ottenuto uno spostamento di 16 bit in una istruzione singola. Sfortunatamente, lo spostamento avviene alla sinistra quando noi vorremmo che avvenisse alla destra. Questo non è un problema.

Concettualmente, MPR può essere spostato a sinistra o a destra. Abbiamo usato un algoritmo di spostamento a destra perchè questo è quello che è usato nell'addizione ordinaria. Comunque, non ha necessariamente bisogno di essere così. L'operazione di addizione è commutativa, e l'ordine può essere invertito: lo spostamento dell'MPR alla sinistra è altrettanto valido. Per approfittare di questo spostamento di 16 bit simulato, dovremo spostare l'MPR alla sinistra. Perciò l'MPR risiederà nel registro H e il risultato nel registro L. Nella Figura 3-25 è mostrata la configurazione di registri che ne risulta.

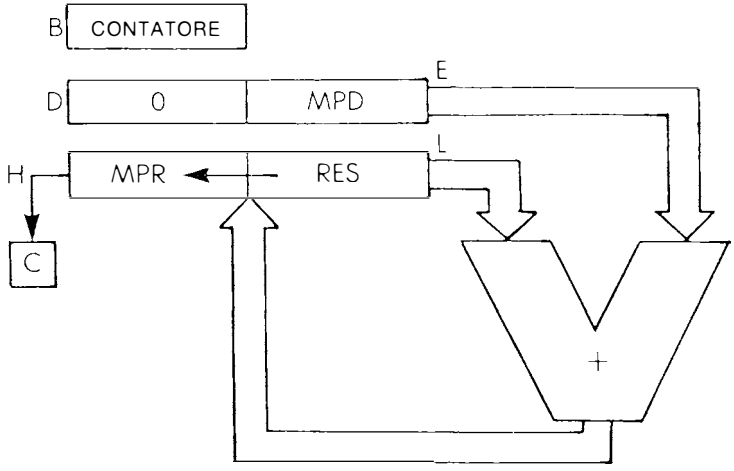


Figura 3-25: Registri per la moltiplicazione migliorata

Il resto del programma è essenzialmente identico al precedente. Ecco il programma che ne risulta.

```
MUL88C   LD      HL, (MPRAD-1)
         LD      L, 0
         LD      DE, (MPDAD)
         LD      D, 0
         LD      B, 8            CONTATORE
MULT     ADD     HL, HL          SCORRIMENTO A SINISTRA
         JR      NC, NOADD
         ADD     HL, DE
NOADD    DJNZ    MULT
         LD      (RESAD), HL
         RET
```

Figura 3-26: Moltiplicazione migliorata, Passo 2

Quando si confronta questo programma col precedente, si può vedere che la lunghezza del loop della moltiplicazione (il numero delle istruzioni tra MULT e il jump) è stato ridotto. Questo programma è stato scritto con meno istruzioni e risulterà usualmente più veloce nell'esecuzione. Ciò mostra il vantaggio di scegliere i registri corretti per contenere le informazioni.

Un progetto semplice darà generalmente origine ad un programma che funziona. Non darà origine ad un programma *ottimizzato*. È perciò importante capire e usare i registri e le istruzioni disponibili nel miglior modo possibile. Questi esempi illustrano un approccio razionale alla scelta dei registri e alla selezione delle istruzioni per la massima efficienza.

Esercizio 3-20: *Calcolate la velocità di un'operazione di moltiplicazione usando quest'ultimo programma. Presumiamo che avvenga un branch nel 50% dei casi. Guardate il numero di cicli richiesti da ogni istruzione nell'indice. Presumiamo una frequenza di clock di 2MHz (un ciclo 2 μs).*

Esercizio 3-21: *Notate che qui abbiamo usato la coppia di registri D ed E per contenere il moltiplicando. Come sarebbe cambiato il programma precedente se avessimo usato invece la coppia di registri B e C? (Consiglio: ciò richiederebbe una modificazione alla fine).*

Esercizio 3-22: *Perché abbiamo dovuto preoccuparci di azzerare il registro D quando si caricava MPD in E?*

Infine, soffermiamoci su un particolare che può apparire irritante al programmatore che non è ancora familiare con lo Z80.

Il lettore avrà notato che, per caricare MPD in E dalla memoria abbiamo dovuto caricare entrambi i registri D ed E allo stesso tempo da un indirizzo della memoria. E questo perché, a meno che l'indirizzo sia contenuto nei registri H ed L, non c'è nessuna maniera di prelevare un byte singolo direttamente e caricarlo nel registro E. Questa è una caratteristica ereditata dal vecchio 8008 il quale non aveva nessun modo di indirizzamento diretto. La caratteristica fu riportata nell'8080, con qualche miglioramento, e migliorata ancor di più nello Z80, dove è possibile prelevare 16 bit direttamente da un dato indirizzo della memoria (ma non 8 bit).

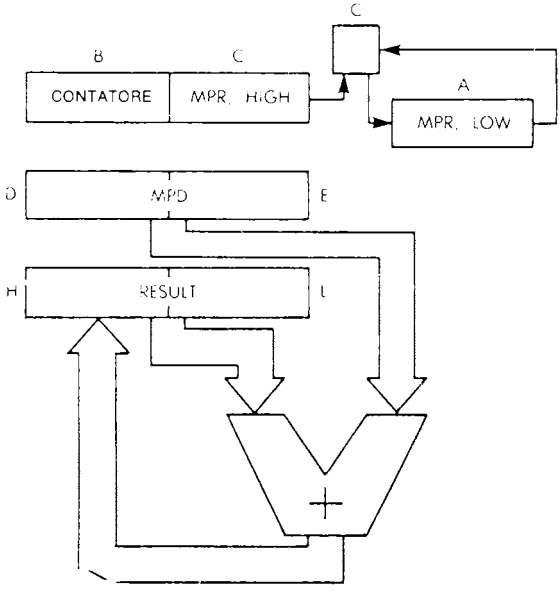


Figura 3-27: Moltiplicazione 16 × 16 — I registri

Ora, avendo risolto questo possibile mistero, eseguiamo una moltiplicazione più complessa.

Una moltiplicazione 16 x 16

Moltiplicheremo due numeri da 16 bit per provare la nostra nuova capacità. Comunque, presumeremo che il risultato richieda solo 16 bit, in modo che possa essere contenuto in una delle coppie dei registri.

Come nel nostro primo esempio di moltiplicazione, il risultato è contenuto nei registri H ed L (vedere la Figura 3-27). Il moltiplicando MPD è contenuto nei registri D ed E.

Si è tentati di depositare un moltiplicatore nel registro B e C. Comunque, se si vuole approfittare dell'istruzione DJNZ, il registro B deve essere destinato al contatore.

Come risultato, metà del moltiplicatore sarà nel registro C, e l'altra metà nel registro A (vedere la Figura 3-27). Il programma della moltiplicazione è il seguente:

```
MUL16   LD      A, (MPRAD+1)MPR, ALTO
        LD      C, A
        LD      A, (MPRAD)  MPR, BASSO
        LD      B, 16       CONTATORE
        LD      DE, (MPDAD) MPD
        LD      HL, 0
MULT    SRL     C           SPOSTAMENTO A DESTRA DELL'MPR ALTO
        RRA                 ROTAZIONE A DESTRA DELL'MPR, BASSO
        JR      NC, NOADD   PROVA IL RIPORTO
        ADD     HL, DE      SOMMA MPD AL RISULTATO
NOADD   EX      DE, HL
        ADD     HL, HL      DOPPIO SPOSTAMENTO DELL'MPD A SINISTRA
        EX      DE, HL
        DJNZ    MULT
        RET
```

Figura 3-28: Programma della moltiplicazione 16 x 16

Il programma è analogo a quelli che abbiamo sviluppato prima. Le prime sei istruzioni (dalla label MUL 16 alla label MULT) eseguono l'inizializzazione dei registri con il contenuto appropriato. Qui è introdotta una complicazione dal fatto che le due metà di MPR devono essere caricate in operazioni separate.

È presupposto che MPRAD punti alla parte bassa dell'MPR nella memoria, seguito dalla parte alta nella successiva posizione di memoria sequenziale. (Notate che può essere usata la convenzione inversa). Una volta che la parte alta di MPR è stata letta in A, deve essere trasferita in C:

```
        LD      A, (MPRAD+1)
        LD      C, A
```

Alla fine, la parte bassa di MPR può essere letta direttamente nell'accumulatore:

```
        LD      A, (MPRAD)
```

Il resto dei registri, B, D, E, H ed L sono inizializzati come al solito:

```
        LD      B, 16
        LD      DE, (MPDAD)
        LD      HL, 0
```

Deve essere eseguito uno spostamento di 16 bit sul moltiplicatore. Richiede due operazioni separate di spostamento e di rotazione sui registri C ed A:

```
MULT    SRL     C
        RRA
```

Dopo lo spostamento di 16 bit, il bit più a destra dell'MPR, cioè, l'LSB, è contenuto nel bit del riporto C dove può essere provato:

```
        JR      NC, NOADD
```

Come al solito, il moltiplicando non è sommato al risultato se il bit del riporto è "0", ed è sommato al risultato se il bit del riporto è "1".

```
        ADD     HL, DE
```

Dopo, il moltiplicando MPD deve essere spostato di una posizione a sinistra.
Comunque, lo Z80 non ha un'istruzione che sposterà il contenuto del registro D ed E simultaneamente a sinistra di una posizione, e non può neanche aggiungere il contenuto di D ed E a sé stesso. Il contenuto di D ed E sarà perciò inizialmente trasferito in H ed L, poi raddoppiato e trasferito indietro a D ed E. Ciò è eseguito dalle successive tre istruzioni:

```
NOADD   EX      DE, HL
        ADD     HL, HL
        EX      DE, HL
```

Alla fine, il contatore B è diminuito ed avviene un jump (salto) all'inizio del loop fino a quando B non diminuisce a "0":

```
        DJNZ    MULT
```

Come al solito, è possibile considerare altre assegnazioni dei registri che possono (o non possono) dar luogo a codici più corti:

Esercizio 3-23: *Caricate il moltiplicatore nei registri B e C. Ponete il contatore in A. Scrivete il corrispondente programma della moltiplicazione e discutete i vantaggi e gli svantaggi di questa assegnazione dei registri.*

Esercizio 3-24: *Riferendoci al programma originale della moltiplicazione a 16 bit della Figura 3-28, potete proporre un modo per spostare l'MPD, contenuto nei registri D ed E, senza trasferirlo nei registri H e L?*

Esercizio 3-25: *Scrivete un programma della moltiplicazione 16 per 16 in grado di rivelare il fatto che il risultato abbia più di 16 bit. Questo è un semplice miglioramento del nostro programma fondamentale.*

Esercizio 3-26: *Scrivete un programma della moltiplicazione 16 × 16 con un risultato di 32 bit. L'assegnazione del registro suggerita appare nella Figura 3-29. Ricordatevi che il risultato iniziale dopo la prima addizione nel loop richiederà solo 16 bit, e che il moltiplicatore libererà un bit per ogni ripetizione susseguente.*

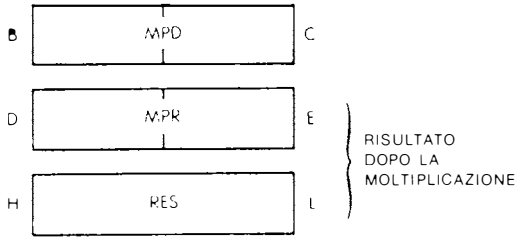


Figura 3-29: Moltiplicazione 16 x 16 con risultato di 32 bit

Adesso esaminiamo l'ultima comune operazione aritmetica, la divisione.

DIVISIONE BINARIA

L'algoritmo per la divisione binaria è analogo a quello che è stato usato per la moltiplicazione. Il divisore è successivamente sottratto dai bit di alto ordine del dividendo. Dopo ogni sottrazione, il risultato è usato invece del dividendo iniziale. Simultaneamente il valore del quoziente è aumentato di 1 ogni volta. Alla fine, il risultato della sottrazione sarà negativo. Questo è chiamato "overdraw". Si deve allora ristabilire il risultato parziale aggiungendogli di nuovo il divisore. Naturalmente il quoziente deve essere simultaneamente diminuito di 1. Il quoziente e il dividendo sono poi spostati di una posizione di bit alla sinistra e l'algoritmo è ripetuto. Il flow-chart (diagramma di flusso) è mostrato nella Figura 3-30.

Il metodo appena descritto è chiamato il *metodo di "restoring"*. Una variante di questo metodo è chiamato il metodo di *non restoring*.

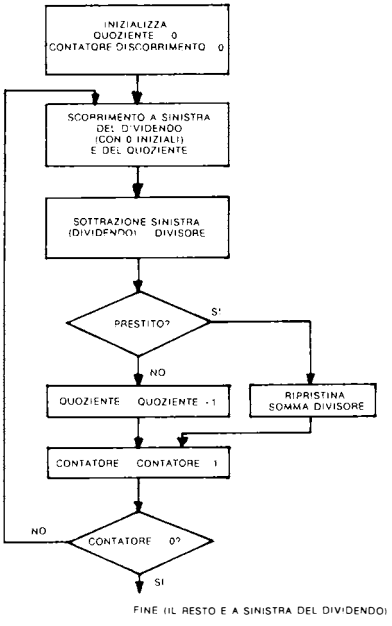


Figura 3-30: Diagramma di flusso della divisione binaria ad 8 bit

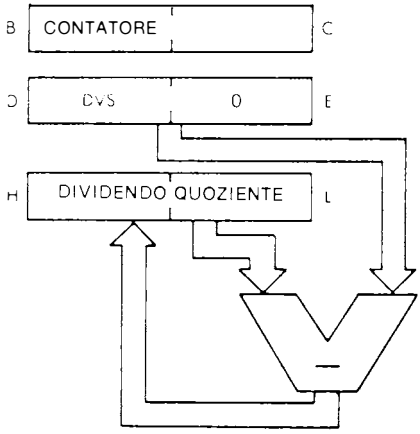


Figura 3-31: Divisione 16 × 8 — I registri

Divisione 16 per 8

Come esempio, esaminiamo qui una divisione 16 per 8, la quale produrrà un quoziente di 8 bit ed un resto di 8 bit. Nella Figura 3-31 è mostrata l'assegnazione dei registri.

Il programma compare di seguito:

```
DIV168   LD    A, (DVSAD)   CARICA IL DIVISORE
         LD    D, A         IN D
         LD    E, 0
         LD    HL, (DVDAD)  CARICA IL DIVIDENDO A 16 BIT
         LD    B, 8         INIZIALIZZA IL CONTATORE
DIV      XOR   A            CANCELLA IL BIT C
         SBC   HL, DE       DIVIDENDO – DIVISORE
         INC   HL           QUOZIENTE = QUOZIENTE + 1
         JP    P, NOADD     PROVA SE IL RESTO È POSITIVO
         ADD   HL, DE       RIPRISTINA SE È NECESSARIO
         DEC   HL           QUOZIENTE = QUOZIENTE – 1
NOADD    ADD   HL, HL       SPOSTA IL DIVIDENDO A SINISTRA
         DJNZ  DIV          RIPETI FINO A CHE B = 0
         RET
```

Figura 3-32: Programma della divisione 16 × 8

Le prime cinque istruzioni nel programma caricano il divisore e il dividendo rispettivamente nei registri appropriati. Inizializzano anche il contatore, nel registro B, al valore 8. Notate di nuovo che il registro B è una posizione preferita per un contatore se deve essere usata l'istruzione specializzata dello Z80 DJNZ:

```
DIV 168  LD    A, (DVSAD)
         LD    D, A
         LD    E, 0
         LD    HL, (DVDAD)
         LD    B, 8
```

Quindi il divisore viene sottratto dal dividendo. Dal momento in cui deve essere usata una istruzione SBC (non c'è nessuna sottrazione a 16 bit senza riporto), il riporto deve essere posto al valore "0" prima di sottrarre. Ciò può essere compiuto in molti modi.

Il riporto può essere azzerato eseguendo istruzioni come:

```
        XOR A
        AND A
        OR A
```

Qui, è usato uno XOR:

```
DIV     XOR     A
```

La sottrazione può allora essere eseguita:

```
        SBC     HL, DE
```

È anticipato che la sottrazione avrà buon esito, cioè, che il resto sarà positivo. Questa è chiamata la fase della "sottrazione di prova", "trial subtract", (riferitevi al flowchart della Figura 3-30). Il quoziente è perciò aumentato di uno. Se la sottrazione non è invece riuscita (cioè, se il resto è negativo), il quoziente dovrà essere diminuito più avanti:

```
        INC     HL
```

Il risultato della sottrazione è poi provato:

```
        JP      P, NOADD
```

Se il resto è positivo o zero, la sottrazione è riuscita, e non è necessario immagazzinarlo. Il programma salta all'indirizzo NOADD. Altrimenti, il dividendo corrente deve essere rimesso al suo valore precedente, aggiungendo nuovamente ad esso il divisore, e il quoziente deve essere diminuito di uno. Questo è eseguito dalle prossime istruzioni:

```
        ADD     HL, DE
        DEC     HL
```

Alla fine, il dividendo che ne risulta è spostato a sinistra, in previsione della prossima operazione di sottrazione di prova. Finalmente, il contatore B è diminuito e provato per il valore "0". Questo "Loop" è eseguito fino a quando B non è zero:

```
NOADD   ADD     HL, HL
        DJNZ    DIV
        RET
```

Esercizio 3-27: *Verificate l'operazione di questo programma di divisione, compilando la tabella della Figura 3-33, come nell'esercizio 3-18 per la moltiplicazione. Notate che il contenuto di D non ha bisogno di essere immesso nella tabella della Figura 3-33, poiché non è mai modificato.*

LABEL	ISTRUZIONE	B	H	

Figura 3-33: Tabella per il Programma della divisione

Divisione ad 8 bit

Il seguente programma usa un metodo di rimemorizzazione e fornisce un quoziente complementato in A. Divide 8 bit per 8 bit (senza segno).

```
  E È IL DIVIDENDO
  C È IL DIVISORE
  A È IL QUOZIENTE
  B È IL RESTO
DIV88     XOR     A          AZZERA ACCUMULATORE
          LD      B, 8       CONTATORE DI LOOP
LOOP88    RL      E          RUOTA CY
          RLA                CY SARÀ OFF
          SUB     C          PROVA DI SOTTRAZIONE DEL DIVISORE
          JP      NC, $+3    SOTTRAZIONE O.K.
          ADD     A, C       RIPRISTINA ACCUMULATORE, PONI CY AD 1
          DJNZ    LOOP88
          LD      B, A       METTI RESTO IN B
          LD      A, E       ACCETTA QUOZIENTE
          RLA                SPOSTA NELL'ULTIMO BIT DEL RISULTATO
          CPL                COMPLEMENTA I BIT
          RET
```

Notate: il simbolo $ nella sesta istruzione rappresenta il valore del contatore di programma.

Divisione non Restoring

Il seguente programma esegue una divisione di numero intero da 16 bit per 16 bit, usando una tecnica che non rimemorizza (non restoring). IX punta al dividendo, IY al divisore (non zero). L'indirizzo che ne risulta è lasciato in IX (vedere la Figura 3-34).

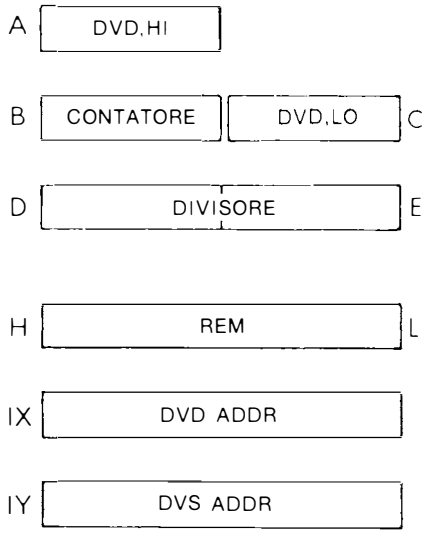


Figura 3-34: Divisore che non rimemorizza — I registri

Il registro B è usato come contatore, inizialmente posto a 16.
A e C contengono il dividendo
D e E contengono il divisore
H e L contengono il risultato
Il dividendo a 16 bit è spostato a sinistra da:
RL C
RLA
Il resto è spostato a sinistra da
ADC HL, HL
Il quoziente finale è lasciato in B, C, con il resto in HL. Il programma è il seguente:

```
DIV16     LD      B, (IX+1)
          LD      C, (IX)
          LD      D, (IY+1)
          LD      E, (IY)
          LD      A, D
          OR      E              (DIVISORE) ALTO OR
                                 (DIVISORE) BASSO
          JR      Z, ERRORE      CECK PER DIVISORE ZERO
          LD      A, B           ACCETTA (DVD) ALTO
          LD      HL, 0          AZZERA RISULTATO
          LD      B, 16          CONTATORE
TRIALSB   RL      C              RUOTA RISULTATO + ACC. A SINISTRA
          RLA
          ADC     HL, HL         SPOSTAMENTO SINISTRA. NON METTE
                                 IL RIPORTO A 1
          SBC     HL, DE         SOTTRAI DIVISORE
NULL      CCF                    BIT DEL RISULTATO
          JR      NC, NGV        ACCUMULATORE NEGATIVO?
```

```
PTV      DJNZ   TRIALSB    CONTATORE ZERO?
         JP     DONE
RESTOR   RL     C          RUOTA RISULTATO + ACC. A SINISTRA
         RLA
         ADC    HL, HL     COME SOPRA
         AND    A
         ADC    HL, DE     RISTABILISCI AGGIUNGENDO DVSR
         JR     C, PTV     RISULTATO POSITIVO
         JR     Z, NULL    RISULTATO ZERO
NGV      DJNZ   RESTOR     CONTATORE ZERO?
DONE     RL     C          SPOSTAMENTO NEL BIT DEL RISULTATO
         RLA
         ADD    HL, DE     RESTO CORRETTO
         LD     B, A       IL QUOZIENTE È IN B, C
         RET
```

Esercizio 3-28: *Paragonate il precedente programma a quello che segue, usando una tecnica di restoring (rimemorizzazione):*

```
DIVIDENDO IN AC
DIVISORE IN DE
QUOZIENTE IN AC
RESTO IN HL

DIV16    LD     HL, 0      AZZERA ACCUMULATORE
         LD     B, 16      PONI CONTATORE
LOOP16   RL     C          ROT ACC-RISULTATO A SINISTRA
         RLA
         ADC    HL, HL     SPOSTAMENTO A SINISTRA
         SBC    HL, DE     SOTTRAZ. DI PROVA DEL DIVISORE
         JR     NC $+3     SOTTRAZIONE
         ADD    HL, DE     RIMEMORIZZAZIONE ACCUM.
         CCF               CALCOLA BIT DEL RISULTATO
         DJNZ   LOOP16     CONTATORE NON ZERO
         RL     C          SPOSTAMENTO NELL'ULTIMO BIT DEL RISULTATO
         RLA
         RET
```

Notate: il simbolo $ significa "locazione corrente" (ottava istruzione).

OPERAZIONI LOGICHE

L'altro tipo di istruzioni che possono essere eseguite dall'ALU all'interno del microprocessore è la serie di *istruzioni logiche*. Esse includono: AND, OR e l'OR esclusivo (XOR). In più, si possono anche includere qui le operazioni di spostamento e di rotazione che sono già state utilizzate e l'istruzione di paragone, chiamata CP per lo Z80. L'uso individuale di AND, OR, XOR, sarà descritto nel capitolo 4 dedicato al set di istruzioni.

Sviluppiamo adesso un breve programma il quale controllerà se una data posizione di memoria chiama LOC contiene il valore "0", il valore "1", o qualcos'altro.

Il programma introdurrà l'istruzione di paragone, e compierà una serie di test logici. Sarà eseguito un segmento del programma o un altro a seconda del risultato del paragone.

Il programma è il seguente:

```
            LD      A, (LOC)      LEGGI IL CARATTERE IN LOC
            CP      00H           PARAGONA CON ZERO
            JP      Z, ZERO       È UNO 0?
            CP      01H           PARAGONA CON UNO
            JP      Z, ONE
ONEFOUND

ZERO

ONE
```

La prima istruzione: "LDA, (LOC)" legge il contenuto della posizione di memoria LOC e lo carica nell'accumulatore. Questo è il carattere che vogliamo provare. È paragonato al valore 0 della seguente istruzione:

```
            CP      00H
```

Questa istruzione paragona il contenuto dell'accumulatore al valore esadecimale "00", cioè, la serie di bit "00000000". Questa istruzione di paragone porrà il bit Z nel registro dei flag al valore "1", se il paragone riesce.

Questo bit può allora essere provato dalla prossima istruzione:

```
            JP      Z, ZERO
```

L'istruzione JUMP prova il valore del bit Z. Se il paragone riesce, il bit Z è stato posto ad uno, e riuscirà il JUMP (il salto). Il programma allora salterà all'indirizzo ZERO. Se il test non riesce, allora sarà eseguita la prossima istruzione sequenziale:

```
            CP      01H
```

Allo stesso modo, la seguente istruzione JUMP salterà alla posizione ONE se riesce il paragone. Se non riesce nessuno dei paragoni, allora sarà eseguita l'istruzione alla posizione NONE FOUND.

```
            JP      Z, ONE
NONEFOUND ...
```

Questo programma è stato introdotto per dimostrare il valore di una istruzione di paragone seguita da un jump. Questa combinazione sarà usata in molti dei programmi che seguono.

Esercizio 3-29: *Riferitevi alle definizioni delle istruzioni LDA, (LOC) nel prossimo capitolo. Esaminate gli effetti di questa istruzione sui flag, se ce ne sono. È necessaria la seconda istruzione di questo programma (CP00H)?*

Esercizio 3-30: *Scrivete il programma che legga il contenuto della posizione di memoria "24" ad un indirizzo chiamato "STAR" se c'era un "*" nella posizione di memoria 24. La serie di bit per un "*" nella numerazione binaria si presume che sia rappresentato da "00101010".*

SOMMARIO DELLE ISTRUZIONI

Usandole, abbiamo ora studiato la maggior parte delle istruzioni importanti dello Z80. Abbiamo trasferito i valori tra la memoria ed i registri. Abbiamo eseguito operazioni aritmetiche e lo-

giche su tali dati. Li abbiamo provati, ed a seconda dei risultati di questi test, abbiamo eseguito varie porzioni del programma.

In particolare, sono state usate speciali istruzioni dello Z80 "automatizzate" come il DJNZ per accorciare i programmi. Altre istruzioni automatizzate: LDDR, CPIR, INIR saranno introdotte nel seguito di questo libro.

È stato fatto un uso completo delle caratteristiche speciali dello Z80, come le istruzioni per registri a 16 bit per semplificare i programmi, e il lettore dovrebbe essere attento a non usare questi programmi su un 8080: sono stati ottimizzati per lo Z80.

Abbiamo anche introdotto una struttura chiamata loop. Ora sarà introdotta un'altra importante struttura di programmazione: la subroutine (sottoprogramma).

LE SUBROUTINE

In concetto, una subroutine o sottoprogramma è soltanto un blocco di istruzioni a cui è stato dato un nome dal programmatore. Da un punto di vista pratico, una subroutine deve iniziare con una istruzione speciale chiamata *subroutine declaration* (dichiarazione della subroutine) che l'identifica come tale per l'assemblatore. Termina anche con un'altra istruzione speciale chiamata *return*. Per prima cosa, illustriamo l'uso di una subroutine in un programma per dimostrare il suo valore. Poi esamineremo com'è veramente implementata.

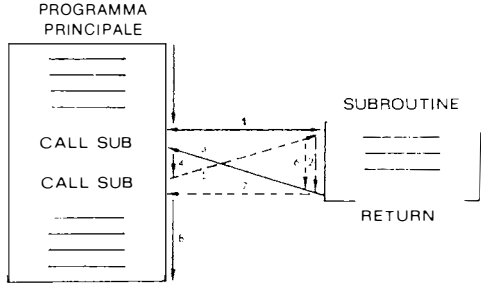


Figura 3-35: Chiamate di subroutine

Nella Figura 3-35 è illustrato l'uso di una subroutine. Il programma principale compare alla sinistra dell'illustrazione.

La subroutine è mostrata simbolicamente alla destra. Esaminiamo il meccanismo della subroutine. Le linee del programma principale sono eseguite successivamente fino a quando è incontrata una nuova istruzione "CALL SUB". Questa istruzione speciale è la *subroutine call* (chiamata del sottoprogramma) e richiede il trasferimento al sottoprogramma. Ciò significa che la prossima istruzione che deve essere eseguita dopo il CALL SUB è la prima istruzione della subroutine. Questa è illustrata dalla freccia 1 sull'illustrazione.

Poi, la subroutine è eseguita proprio come ogni altro programma. Presumeremo che la subroutine non contenga nessun'altra chiamata. L'ultima istruzione di questa subroutine è un RETURN. Questa è una istruzione speciale che provocherà un ritorno al programma principale. La prossima istruzione che deve essere eseguita dopo il RETURN è quella che segue il CALL SUB nel programma principale. Ciò è illustrato dalla freccia 3 sull'illustrazione. L'esecuzione del programma poi continua, come illustrato dalla freccia 4. Nel corpo del programma principale compare una seconda CALL SUB. Avviene un nuovo trasferimento, mostrato dalla freccia 5. Questo significa che il corpo della subroutine è nuovamente eseguito seguendo l'istruzione CALL SUB. Ogni qual volta è incontrato il RETURN all'interno della subroutine, avviene un ritorno all'istruzione che segue la CALL SUB in questione. Ciò è illustrato dalla freccia 7. Se-

guendo il ritorno al programma principale, l'esecuzione del programma procede normalmente, come illustrato dalla freccia 8. L'effetto delle due istruzioni speciali CALL SUB e RETURN ora dovrebbe essere chiaro. Qual è il valore del meccanismo della subroutine? Il valore essenziale della subroutine è che essa può essere chiamata da un numero qualunque di punti del programma principale, ed usata ripetutamente *senza riscriverla*.

Un primo vantaggio è quello che questo approccio risparmia spazio di memoria. poiché non c'è nessun bisogno di riscrivere ogni volta la subroutine. Un secondo vantaggio è che il programmatore può progettare una subroutine specifica solo una volta e poi ripeterla. Questa è una semplificazione significativa del progetto del programma.

Esercizio 3-31: *Qual è il principale svantaggio di una subroutine?* (Segue la risposta).

Lo svantaggio della subroutine dovrebbe essere chiaro esaminando solo il flusso di esecuzione tra il programma principale e la subroutine. Una subroutine genera una *esecuzione più lenta*, poiché devono essere eseguite istruzioni extra: la CALL SUB e il RETURN.

Realizzazione del meccanismo della subroutine

Qui esamineremo come le due istruzioni speciali, CALL SUB e RETURN, sono implementate internamente al processore. L'effetto dell'istruzione CALL SUB è quello di fare sì che l'istruzione successiva sia prelevata ad un nuovo indirizzo. Vi ricorderete che l'indirizzo dell'istruzione successiva che deve essere eseguita in un computer è contenuto nel program counter (PC).

Ciò significa che l'effetto della CALL SUB è quello di sostituire il nuovo contenuto nel registro PC. Il suo effetto è quello di caricare l'indirizzo dell'inizio della subroutine nel program counter. *Ciò è veramente sufficiente?* Per rispondere a questa domanda consideriamo l'altra istruzione che deve essere realizzata: Il RETURN. Il RETURN deve provocare, come indica il suo nome, un ritorno all'istruzione che segue la CALL SUB. Ciò è possibile solo se l'indirizzo di questa istruzione è stato conservato da qualche parte. In realtà questo indirizzo è pari al valore del PC al momento di incontrare la CALL SUB. Questo perché il contatore del programma è automaticamente aumentato ogni volta che è usato (rileggere il capitolo 1). Questo è esattamente l'indirizzo che vogliamo conservare, in modo che più tardi possiamo eseguire il RETURN.

Il prossimo problema è: dove possiamo conservare questo indirizzo del ritorno? Questo indirizzo deve essere conservato in una posizione dove è garantito che non sia cancellato.

Comunque, consideriamo adesso l'istruzione seguente illustrata dalla Figura 3-36. In questo esempio, la subroutine 1 contiene una chiamata alla SUB 2.

Il nostro meccanismo dovrebbe funzionare anche in questo caso. Naturalmente, ci potrebbero anche essere più di due subroutine, diciamo per esempio N chiamate "annidate" una dentro l'altra (sono dette "nested calls").

Ogni qual volta s'incontra una nuova CALL (chiamata), il meccanismo deve perciò immagazzinare di nuovo il PC. Questo implica che per questo meccanismo abbiamo almeno bisogno di 2 N posizioni di memoria. Addizionalmente, avremo bisogno di ritornare per prima cosa da SUB 2 e poi da SUB 1. In altre parole, abbiamo bisogno di una struttura che possa conservare l'ordine cronologico in cui sono stati messi in serbo gli indirizzi.

La struttura ha un nome ed è già stata introdotta. È lo *stack*. La Figura 3-38 mostra il vero contenuto dello stack durante le chiamate successive delle subroutine. Consideriamo inizialmente il programma principale. All'indirizzo 100, si incontra la prima chiamata: CALL SUB 1. Presumeremo che, in questo microprocessore, la chiamata della subroutine impieghi 3 byte (RST è un'eccezione). Perciò il prossimo indirizzo sequenziale non è "101", ma "103". L'istruzione CALL usa gli indirizzi "100", "101", "102". Siccome l'unità di controllo dello Z80 "sa" che è un'istruzione di 3 byte, il valore del contatore di programma sarà "103" quando la chiamata è stata completamente decodificata. Il risultato della chiamata sarà di caricare il valore "280" nel contatore di programma. Il "280" è l'indirizzo di inizio di SUB 1.

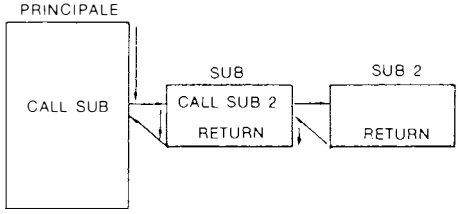


Figura 3-36: Chiamate annidate

Adesso siamo pronti a dimostrare il risultato dell'istruzione di RETURN e il corretto funzionamento del nostro meccanismo di stack. L'esecuzione procede entro la SUB 2 fino a che si incontra l'istruzione RETURN al tempo 3. Il risultato dell'istruzione RETURN è semplicemente quella di trasferire il contenuto della cima dello stack nel program counter. In altre parole, il contatore di programma è ristabilito al suo valore precedente all'entrata nella subroutine. La cima dello stack nel nostro esempio è "303". La Figura 3-38 mostra che, al tempo 3, il valore "303" è stato rimosso dallo stack ed è stato rimesso nel program counter. Come risultato, l'esecuzione dell'istruzione procede dall'indirizzo "303". Al tempo 4 si incontra il RETURN di SUB 1. Il valore sulla cima dello stack è "103". È prelevato ed è caricato nel contatore di programma. Come risultato, l'esecuzione del programma procederà dalla posizione "103" in avanti entro il programma principale.

Questo è, veramente l'effetto che noi volevamo. La Figura 3-38 mostra che al tempo 4 lo stack è vuoto nuovamente. Il meccanismo funziona.

Il meccanismo di chiamata della subroutine funziona fino alla dimensione massima dello stack. Ecco perché i vecchi microprocessori che avevano uno stack da 4 o da 8 locazioni erano essenzialmente limitati a 4 o 8 livelli di chiamate di subroutine.

Notate che, nelle Figure 3-36 e 3-37, le subroutine sono state mostrate alla destra del programma principale. In realtà, le subroutine sono battute da chi le usa come istruzioni regolari del programma. Su un foglio di carta, quando si presenta l'elencazione del programma completo, le subroutine possono essere all'inizio del testo, a metà, o alla fine. Ecco perché sono precedute da una dichiarazione di subroutine: devono essere identificate.

Le istruzioni speciali dicono all'assemblatore che quello che segue dovrebbe essere trattato come una subroutine.

Tali *direttive* per l'assemblatore saranno discusse nel Capitolo 10.

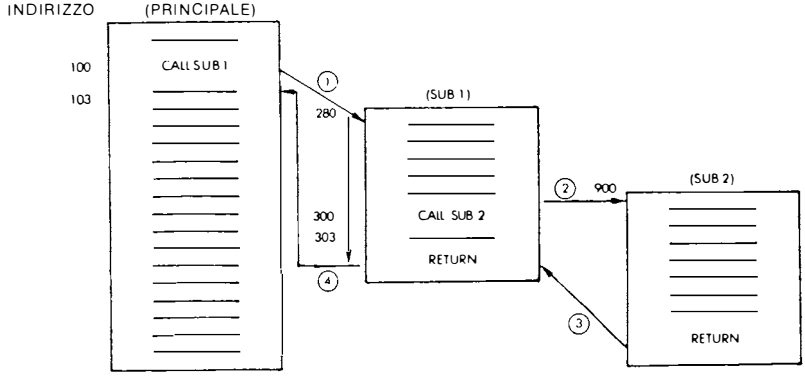


Figura 3-37: Chiamate di subroutine

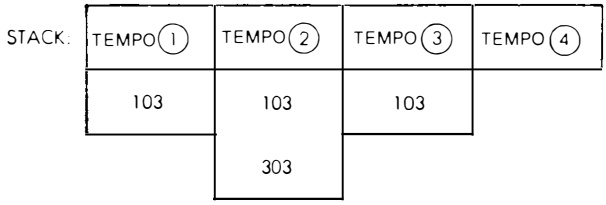


Figura 3-38: Contenuto dello stack in funzione del tempo

Subroutine per lo Z80

I concetti fondamentali relativi alle subroutine sono adesso stati presentati. È stato mostrato che lo stack è richiesto per rendere effettivo questo meccanismo. Lo Z80 è fornito di un registro puntatore dello stack a 16 bit. Perciò, lo stack può risiedere ovunque all'interno della memoria e può avere fino a 64 K (1K = 1024) byte, presumendo che essi siano disponibili per quello scopo. In pratica, l'indirizzo di inizio dello stack, come la sua dimensione massima, sarà definito dal programmatore prima di scrivere il suo programma. Allora allo stack sarà riservata un'area di memoria.

L'istruzione di chiamata della subroutine, nel caso dello Z80, è chiamata CALL, e avviene in due versioni; la chiamata diretta o incondizionale, come la CALL ADDRESS, è quella che abbiamo già descritto. In più, lo Z80 è fornito di una istruzione di chiamata condizionale che chiamerà una subroutine se è soddisfatta una condizione. Per esempio: CALL NZ, SUB 1 risulterà in una chiamata alla subroutine 1 se il risultato della precedente operazione non è zero.

Questo è una potente "facility", dal momento in cui molte chiamate di subroutine sono condizionali, cioè: avvengono solo se si incontra una qualche condizione specifica.

CALL CC, NN è eseguita solo se è vera la condizione specificata da "CC". CC è una serie di tre bit (i bit 4, 5 e 6 del codice operativo) che può specificare fino a otto condizioni. Corrispondono rispettivamente ai quattro flag "Z", "C", "P/V", "S" che sono a zero o non a zero.

Allo stesso modo, sono fornite due tipi di istruzioni di ritorno: RET e RET CC.

RET è l'istruzione di ritorno fondamentale. Occupa un byte e fa si che i due byte più in alto dello stack siano re-installati nel program counter. È incondizionale.

RET CC ha lo stesso effetto eccetto che è eseguito solo se sono vere le condizioni specificate da CC. I bit della condizione sono gli stessi che per l'istruzione CALL appena descritta.

Addizionalmente, sono disponibili due tipi specializzati di ritorno, i quali sono usati per terminare le routine d'interrupt: RETI, RETN. Sono descritti nel paragrafo sulle istruzioni dello Z80 cosi come in quello sugli interrupt.

In fine, è fornita un'altra istruzione specializzata che è analoga alla chiamata della subroutine, ma permette al programma di saltare ad una sola delle otto posizioni di inizio localizzate nella pagina zero. Questa è l'istruzione RST P. Questa è un'istruzione ad un byte che conserva automaticamente il contatore di programma nello stack, e provoca un salto all'indirizzo da tre bit specificato dal campo P. Il campo P corrisponde ai bit 4, 5 e 6 dell'istruzione, moltiplicati per otto.

In altre parole, se i bit 4, 5 e 6 sono "000", il salto avverrà alla posizione 00H. Se questi bit sono "001", la biforcazione avverrà a 08H, ecc., fino a 111, che causerà una biforcazione alla posizione 38H. l'istruzione RST è molto efficiente in termini di velocità poichè è un'istruzione da un byte singolo. Comunque, può saltare a solo otto posizioni, nella pagina 0. Inoltre, questi indirizzi in pagina 0 sono distanti soltanto otto byte. Questa istruzione è un'eredità dell'8080 ed era ampiamente usata per gli interrupt. Ciò sarà descritto nella sezione degli interrupt. Comunque, questa istruzione può essere usata per qualsiasi altro scopo dal programmatore, e dovrebbe essere considerata come una possibile chiamata di subroutine specializzata.

Esempi di Subroutine

La maggior parte dei programmi che abbiamo sviluppato e che svilupperemo dovrebbero essere generalmente scritti come subroutine.

Per esempio, il programma della moltiplicazione è probabile che sia usato da molte aree del programma. Per facilitare e chiarire lo sviluppo del programma, è perciò conveniente definire una subroutine il cui nome sarebbe, per esempio, MULT. Alla fine di questa subroutine aggiungeremmo semplicemente l'istruzione RET.

Esercizio 3-32: *Se è usato MULT come subroutine, potrebbe "danneggiare" qualche flag interno o registro?*

Recursione

Recursione è una parola usata per indicare che una subroutine sta chiamando sè stessa. Se avete capito il meccanismo di esecuzione, ora dovreste esser capaci di rispondere alla seguente domanda:

Esercizio 3-33: *È lecito lasciare che una subroutine chiami sé stessa? (In altre parole, funzionerà ogni cosa anche se una subroutine chiama sé stessa?) Se non siete sicuri, svuotate lo stack e riempitelo con gli indirizzi successivi. Poi, guardate i registri e la memoria (vedere l'Esercizio 3-18) e determinate se esiste il problema.*

Gli interrupt saranno trattati nel capitolo dell'input output (capitolo 6). Tutti i ritorni sono istruzioni di un byte; tutte le chiamate sono istruzioni di 3 byte (eccetto RST).

Esercizio 3-34: *Nel prossimo capitolo, guardate i tempi di esecuzione delle istruzioni CALL e RET. Perchè il ritorno da una subroutine è tanto più veloce della CALL? (Suggerimento: se la risposta non è ovvia, riguardate alla realizzazione con lo stack del meccanismo della subroutine e analizzate le operazioni interne che devono essere eseguite.*

Parametri della Subroutine

Quando chiamate una subroutine, normalmente ci si aspetta che essa lavori su alcuni dati. Per esempio, nel caso della moltiplicazione, si vuole trasmettere due numeri alla subroutine che eseguirà la moltiplicazione. Abbiamo visto nel caso del programma della moltiplicazione che questa subroutine si aspettava di trovare il moltiplicatore e il moltiplicando in date posizioni di memoria. Questo illustra un metodo di passaggio dei parametri: attraverso la memoria. Sono usate altre due tecniche, cosi che abbiamo tre modi di passaggio di parametri:

1 – attraverso i registri
2 – attraverso la memoria
3 – attraverso lo stack

I *registri* possono essere usati per passare i parametri. Questa è una soluzione vantaggiosa, purché siano disponibili i registri, poichè non si ha bisogno di usare una posizione di memoria fissa: la subroutine rimane indipendente nella memoria. Se è usata una posizione di memoria fissa, chiunque usa la subroutine deve stare molto attento ad usare la stessa convenzione e che la posizione di memoria sia veramente disponibile (guardate all'esercizio 3-19). Ecco perchè, in molti casi, un blocco di posizioni di memoria è riservato semplicemente al passaggio dei parametri tra le varie subroutine.

Usare la memoria ha il vantaggio di una più grande flessibilità (più dati), ma risulta in prestazioni più scarse ed il collegamento della subroutine ad una data area di memoria.

Il depositare parametri nello *stack* ha lo stesso vantaggio dell'usare i registri: è indipendente dalla memoria.

La subroutine semplicemente sa che deve ricevere, diciamo, due parametri che sono immagazzinati nella cima dello stack. Naturalmente ha degli svantaggi: occupa lo stack con dati e,

perciò, riduce il numero di possibili livelli di chiamate di subroutine. Complica anche in modo significativo l'uso dello stack, e può richiedere stack multipli.

La scelta sta al programmatore. In generale si desidera rimanere indipendenti quanto più possibile dalle varie posizioni di memoria.

Se i registri non sono disponibili, lo stack è una possibile soluzione. Comunque, queste informazioni possono dover risiedere direttamente nella memoria se una grande quantità di informazioni dovesse essere passata ad una subroutine. Un modo elegante per risolvere il problema di passare un blocco di dati è semplicemente quello di trasmettere un puntatore come informazione. Un *puntatore* è l'indirizzo dell'inizio del blocco. Un puntatore può essere trasmesso in un registro, o nello stack (per immagazzinare un indirizzo da 16 bit possono essere usate due locazioni dello stack), o in una data locazione (o locazioni) di memoria.

Alla fine, se non è applicabile nessuna delle due soluzioni, allora si può fare un compromesso con la subroutine cosicchè i dati saranno in una qualche posizione di memoria fissa (la "cassetta della posta").

Esercizio 3-35: *Quale dei tre metodi detti sopra è il migliore per la recursione?*

Biblioteca di Subroutine

C'è un forte vantaggio nello strutturare porzioni di un programma in subroutine identificabili: esse possono essere spulciate (debugged) dagli errori indipendentemente e possono avere un nome mnemonico. Diventano divisibili purché siano usate in altre aree del programma, e cosi ci si può costruire una biblioteca di subroutine utili. Comunque, nella programmazione del computer non c'è nessuna panacea generale. L'usare le subroutine in modo sistematico per ogni gruppo di istruzioni che possono essere raggruppate per funzione può anche risultare in scarsa efficienza. Il programmatore attento dovrà pesare i vantaggi e gli svantaggi.

SOMMARIO

Questo capitolo ha presentato il modo in cui sono manipolate le informazioni all'interno dello Z80 tramite le istruzioni. Sono stati introdotti algoritmi sempre più complessi e tradotti in programmi. Sono stati usati e spiegati i principali tipi di istruzioni.

Sono state definite importanti strutture come i loop, gli stack e le subroutine.

Adesso voi dovreste avere acquisito una comprensione fondamentale della programmazione e le maggiori tecniche usate nelle applicazioni standard.

Studiamo le istruzioni disponibili.

```
    A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD    BC,(0200)
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0200)
    A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD    B,08
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106  LD    DE,(0202)
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0202)
    A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A  LD    B,00
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C  LD    HL,0000
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0000)
    A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL   C
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
C   A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111  JR    NC,0114
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114)
C   A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD   HL,DE
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA   E
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
V   A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116  RL    D
   A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Figura 3-39: Moltiplicazione: Una trace (traccia) completa

```
Z V     A=00 BC=0801 DE=000A HL=0005 S=0300 P=0119 0119' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N    A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP   NZ,0101
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0101')
   N    A=00 BC=0701 DE=000A HL=0005 S=0300 P=0101 0101  SKIP 1
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C   A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0114')
Z V L   A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD  HL,DE
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0700 DE=000A HL=0001 S=0300 P=0114 0114  SLA  1
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V     A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116  RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N    A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP   NZ,0101
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0101')
   N    A=00 BC=0600 DE=0014 HL=000F S=0300 P=0101 0101  SKIP C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0114')
Z V     A=00 BC=0600 DE=0014 HL=0001 S=0300 P=0114 0114  SLA  1
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V     A=00 BC=0600 DE=0028 HL=0001 S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N    A=00 BC=0500 DE=0028 HL=0001 S=0300 P=0119 0119  JP   NZ,0101
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0101')
   N    A=00 BC=0500 DE=0028 HL=0001 S=0300 P=0101 0101  SKIP C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0114')
Z V     A=00 BC=0500 DE=0028 HL=0001 S=0300 P=0114 0114' SLA
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V     A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116  RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0500 DE=0050 HL=0001 S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N    A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119  JP   NZ,0101
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0101')
   N    A=00 BC=0400 DE=0050 HL=000F S=0300 P=0101 0101  SKIP C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0400 DE=0050 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0114')
Z V     A=00 BC=0400 DE=00A0 HL=0001 S=0300 P=0114 0114  SLA  1
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V     A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N    A=00 BC=0300 DE=00A0 HL=0001 S=0300 P=0119 0119  JP   NZ,0101
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0101')
   N    A=00 BC=0300 DE=00A0 HL=0001 S=0300 P=0101 0101' SKIP C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0111 0111  JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0114')
Z V     A=00 BC=0300 DE=00A0 HL=0001 S=0300 P=0114 0114  SLA  1
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  C     A=00 BC=0300 DE=0040 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00 BC=0300 DE=0140 HL=0001 S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N    A=00 BC=0200 DE=0140 HL=000F S=0300 P=0119 0119' JP   NZ,0101
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00     (0101')
   N    A=00 BC=0200 DE=0140 HL=000F S=0300 P=0101 0101' SKIP C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Figura 3-39: Moltiplicazione: Una trace (traccia) completa (continua)

| | | |

LABEL	ISTRUZIONE	B	C	C (RIPORTO)	D	E	H	L
		00	00	0	00	00	00	00
MP488	LD BC,(0200)	00	03	0	00	00	00	00
	LD B, 08	08	03	0	00	00	00	00
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D, 00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	1	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05
MULT	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
	ADD HL,DE	07	00	0	00	0A	00	0F
NO ADD	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

Figura 3-41: Due iterazioni del Loop

CAPITOLO 4

IL SET DI ISTRUZIONI DELLO Z80

INTRODUZIONE

Questo capitolo analizzerà preliminarmente i vari tipi di istruzioni che dovrebbero essere disponibili in un computer general-pourpose (per scopi generali). Poi analizzerà una alla volta tutte le istruzioni disponibili per lo Z80, e spiegherà dettagliatamente il loro scopo e le maniere in cui esse interessano i flag o possono essere usate insieme con i vari modi di indirizzamento. Nel Capitolo 5 sarà presentata una discussione dettagliata delle tecniche d'indirizzamento.

CLASSI DI ISTRUZIONI

Le istruzioni possono essere classificate in molti modi, e non c'è nessuno standard. Noi qui distingueremo cinque principali classi di istruzioni:

1 – trasferimenti dei dati
2 - elaborazione dei dati
3 – test e branch
4 - input/output (ingresso/uscita)
5 – controllo

Ora, esaminiamo ognuna di queste categorie di istruzioni alla volta.

Trasferimento dei Dati

Le istruzioni per il trasferimento dei dati trasferiranno i dati tra i registri, o tra un registro e la memoria, o tra un registro e un dispositivo input/output. Possono esistere istruzioni di trasferimento specializzate per i registri che giocano un ruolo specifico. Per esempio le operazioni di "push" (inserisci) e di "pop" (preleva) sono fornite per un efficiente funzionamento dello stack. Esse sposteranno una parola di dati tra la cima dello stack e l'accumulatore in una singola istruzione, mentre aggiornano automaticamente il registro puntatore dello stack.

Elaborazione Dati

L'istruzione di elaborazione dati si suddivide in cinque categorie generali:

1 – operazioni aritmetiche (come più/meno)
2 – manipolazione dei bit (set e reset)
3 – incremento e decremento
4 – operazioni logiche (come AND, OR, OR esclusivo)
5 – operazioni di skew e shift (come spostare e ruotare)

Si dovrebbe notare che, per l'elaborazione efficiente dei dati, è desiderabile avere potenti istruzioni aritmetiche, come la moltiplicazione e la divisione. Sfortunatamente non sono disponibili sulla maggior parte dei microprocessori.

È anche desiderabile avere potenti istruzioni di shift (spostamento) e skew (asimmetria), come spostare n bit, o uno scambio di nibble, dove la metà destra e la metà sinistra del byte sono scambiate. Anche questi non sono di solito disponibili sulla maggior parte dei microprocessori.

Prima di esaminare le vere istruzioni dello Z80, richiamiamo alla mente la differenza tra uno *shift* (spostamento) e una *rotation* (rotazione). Lo shift muoverà il contenuto di un registro o una posizione di memoria di una posizione di bit a sinistra o a destra. Il bit che cade fuori dal registro entrerà nel bit di riporto. Il bit che entra sull'altro lato sarà uno "0" eccetto nel caso di uno "shift aritmetico a destra, dove sarà duplicato l'MSB.

Nel caso di una rotazione, il bit che esce va ancora nel riporto. Invece, il bit che entra è il valore precedente che era nel bit del riporto. Ciò corrisponde ad una rotazione a 9 bit. È spesso desiderabile avere una vera rotazione ad 8 bit dove il bit che entra su un lato è quello che cade dall'altro lato. Questo non è fornito sulla maggior parte dei microprocessori ma è disponibile sullo Z80 (vedere la Figura 4-1).

Infine, quando si sposta una parola alla destra, è conveniente avere un ulteriore tipo di shift, chiamato "estensione di segno" oppure "shift aritmetico a destra".

Quando si fanno operazioni sui numeri in complemento a due, particolarmente quando si eseguono routine in virgola mobile, è spesso necessario spostare un numero negativo a destra. Quando spostate a destra un numero in complemento a due, il bit che deve entrare sul lato sinistro dovrebbe essere un "1" (il segno dovrebbe essere ripetuto tutte le volte che è necessario dagli shift successivi). Questo è lo shift aritmetico a destra.

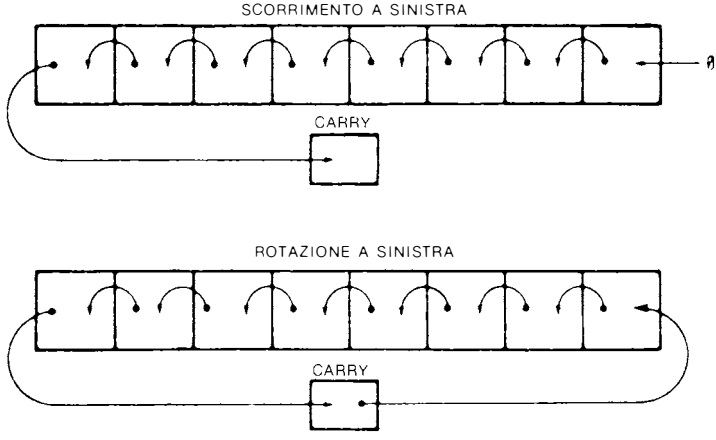


Figura 4-1: Scorrimento e rotazione

Test e Jump

Le istruzioni di test proveranno i bit nel registro specificato per "0" o "1", oppure combinazioni. Al minimo, deve essere possibile provare il registro dei flag. Perciò, è desiderabile avere in questo registro quanti più flag possibili.

Inoltre, è conveniente essere in grado di provare combinazioni di tali bit con una singola istruzione.

Infine è desiderabile essere capaci di provare *ogni posizione di bit in ogni registro*, e di provare il valore di un registro paragonato al valore di ogni altro registro (più grande di, più piccolo

di, uguale). Le istruzioni di prova dei microprocessori sono di solito limitate a provare bit singoli del registro dei flag. Comunque, lo Z80 offre migliori possibilità della maggior parte degli altri microprocessori.

Le istruzioni di jump che possono essere disponibili, generalmente comprendono tre categorie:

1 - il jump, che specifica un indirizzo completo a 16 bit.
2 il salto relativo, che è spesso ristretto ad un campo di spostamento di 8 bit.
3 la chiamata, che è usata con le subroutine.

È conveniente avere due o anche tre modi di salto, a seconda, per esempio, se il risultato di un paragone è "più grande di", "meno grande di" o "uguale". È anche conveniente avere operazioni di skip (salto), le quali salteranno avanti o indietro di qualche istruzione. Comunque, uno "skip" è equivalente ad un "jump". Alla fine, nella maggior parte dei loop, di solito c'è un'operazione di incremento o di decremento, seguita da un test-and-branch. La disponibilità di un incremento/decremento ad istruzione singola più il test-and-branch è, perciò, un vantaggio significativo per la realizzazione di loop efficienti. Questo non è disponibile nella maggior parte dei microprocessori. Sono disponibili soltanto branch semplici uniti con test semplici. Naturalmente, questo complica la programmazione e riduce l'efficienza. Nel caso dello Z80, è disponibile un'istruzione di "decremento e salto". Comunque, essa prova soltanto un registro specifico (B) per zero.

Input/Output

Le istruzioni di input/output sono istruzioni specializzate per il trattamento dei dispositivi di input/output. In pratica, la maggioranza dei microprocessori ad 8 bit usa *memory-mapped I/O* (I/O in mappaggio di memoria): i dispositivi input/output sono collegati al bus degli indirizzi proprio come chip di memoria e indirizzati come tali. Al programmatore essi compaiono come locazioni di memoria. Tutte le operazioni di memoria normalmente richiedono 3 byte e sono, perciò, lente. Per un trattamento di input/output efficiente in tale condizione, è desiderabile avere un breve meccanismo di indirizzamento in modo che i dispositivi di I/O la cui velocità di trattamento è cruciale possano risiedere nella pagina 0.

Comunque, se è disponibile l'indirizzamento in pagina 0, essa è di solito usata per la memoria RAM, cosa che impedisce il suo uso efficace per i dispositivi di input/output. Lo Z80, come l'8080, è fornito di istruzioni di I/O specializzate. Come risultato, nel caso dello Z80, il progettista può usare entrambi i metodi: i dispositivi di input/output possono essere indirizzati come dispositivi della memoria, altrimenti come dispositivi di input/output, usando le istruzioni di I/O.

Esse saranno descritte più avanti in questo capitolo.

Istruzioni di Controllo

Le istruzioni di controllo forniscono segnali di sincronizzazione e possono sospendere o eseguire l'interrupt di un programma. Possono anche funzionare come una sosta o un interrupt simulato. (Gli interrupt saranno descritti nel Capitolo 6 nelle Tecniche di Input/Output.)

IL SET DI ISTRUZIONI DELLO Z80

Introduzione

Il microprocessore Z80 fu progettato come rimpiazzo dell'8080 e per offrire possibilità addizionali. Come risultato di questa filosofia di progettazione, lo Z80 offre tutte le istruzioni dell'8080, più delle istruzioni addizionali. In vista del numero limitato di bit disponibili in un codice operativo di 8 bit, ci si può domandare come i progettisti dello Z80 siano riusciti a rendere disponibili molte istruzioni addizionali. L'hanno fatto usando alcuni codici operativi dell'8080 non usati e aggiungendo un byte addizionale al codice operativo per le operazioni indicizzate. Ecco perchè alcune delle istruzioni dello Z80 occupano fino a cinque byte nella memoria.

È importante ricordare che ogni programma può essere scritto in molti differenti modi. È indispensabile una conoscenza e una comprensione totale del set di istruzioni per raggiungere una programmazione efficiente. Comunque, quando imparate come programmare, non è essenziale scrivere programmi ottimizzati. Durante una prima lettura di questo capitolo non è perciò importante ricordare tutte le varie istruzioni. È importante ricordare le categorie di istruzioni e studiare gli esempi tipici. Poi, nella scrittura dei programmi, il lettore dovrebbe consultare la descrizione del set di istruzioni dello Z80, e scegliere le istruzioni che più si adattano ai suoi bisogni. Le varie istruzioni dello Z80 saranno perciò riviste in questo capitolo con lo scopo di semplificarle e raggrupparle in categorie logiche. Il lettore interessato ad esplorare le capacità delle varie istruzioni è rimandato alle descrizioni individuali delle istruzioni.

Ora esamineremo le capacità fornite dallo Z80 nei termini dei cinque tipi di istruzioni che sono state definite all'inizio di questo capitolo.

Istruzioni per il trasferimento dei dati

Le istruzioni per il trasferimento dei dati sullo Z80 possono essere classificate in quattro categorie: trasferimenti ad 8 bit, trasferimenti a 16 bit, operazioni di stack, e trasferimenti di blocchi. Esaminiamole.

Trasferimenti dei dati ad 8 bit

Tutti i trasferimenti dei dati ad 8 bit sono compiuti tramite istruzioni di carico.
Il formato è:

LD destinazione, sorgente.

Per esempio, l'accumulatore A può essere caricato nel registro B usando le istruzioni:

LD B, A

I trasferimenti diretti possono essere eseguiti tra qualsiasi coppia di registri di lavoro (ABCDEHL). Per caricare qualche registro di lavoro, eccetto per l'accumulatore, da una locazione di memoria, l'indirizzo di questa locazione di memoria deve essere inizialmente caricato in qualche coppia di registri, come i registri H e L.

Per esempio, per caricare il registro C dalla locazione di memoria 1234, il registro H e L dovrà per prima cosa essere caricato con il valore "1234". (Sarà usata una istruzione di carico che opera su 16 bit). Questo è descritto nel prossimo paragrafo. Poi, l'istruzione LDC, (HL) sarà usata e darà il risultato desiderato.

L'accumulatore è una eccezione. Può essere caricato direttamente da ogni posizione di memoria specificata. Questo è chiamato il modo di indirizzamento esteso. Per esempio, per caricare l'accumulatore con il contenuto della locazione di memoria 1234, sarà usata la seguente istruzione:

LD A, (1234H) (Notate l'uso delle "()" per indicare il "contenuto di").

L'istruzione sarà immagazzinata nella memoria come segue:

indirizzo	PC : 3A	(codice operativo)
	PC + 1 : 34	(metà bassa dell'indirizzo)
	PC + 2 : 12	(metà alta dell'indirizzo)

Notate che l'indirizzo è immagazzinato in "ordine inverso" nella stessa istruzione.

3A	indirizzo basso	indirizzo alto

Tutti i registri di lavoro possono anche essere caricati con ogni specificato valore ad otto bit,

o "literall' (letterale), contenuto nel secondo byte dell'istruzione (questo è chiamato indirizzamento immediato). Un esempio è:

LD E, 12H

il quale carica il registro E con il valore 12 esadecimale.
Nella memoria, l'istruzione appare come:

PC	: 1E	(codice operativo)
PC + 1	: 12	(operando "literal")

Come risultato di questa operazione, l'operando immediato o valore literal sarà contenuto nel registro E. È disponibile anche l'*indirizzamento indicizzato* per caricare il contenuto dei registri, e sarà ampiamente descritto nel prossimo capitolo sulle tecniche di indirizzamento. Esistono altre possibilità multiformi per caricare registri specifici, e nella Figura 4-2 è mostrata una tabella che elenca tutte le possibilità (tabelle fornite da Zilog, Inc). Le aree grige mostrano le istruzioni comuni con l'8080 A.

DESTINAZIONE \ SORGENTE		IMPLICATO		REGISTRO							REG INDIRETTO			INDICIZZATO		INDIR 'EST'	IMME
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
REGISTRO	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	7A	DD 7E d	FD 7E d	32 n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	45	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			9F	59	5A	5B	5C	5D	5E	97			DD 5E d	FD 5E d		1E n
	H			60	61	62	63	64	65	66	6F			DD 66 d	FD 66 d		26 n
	L			63	69	64	66	8C	6D	6E				DD 6E d	FD 6E d		2E n
REG INDIRETTO	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			62													
	(DE)			12													
INDICIZ-ZATO	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
INDIR. EST.	(nn)			32 n n													
IMPLI-CATO	I			ED 47													
	R			ED 4F													

Figura 4-2: Istruzioni di caricamento ad 8 bit – 'LD'

Trasferimenti dei dati a 16 bit

Fondamentalmente, ciascuna coppia di registro a 16 bit, BC, DE, HL, SP, IX, IY, possono essere caricate con un operando letterale a 16 bit, o da uno specificato indirizzo di memoria (*indirizzamento esteso*), o dalla cima dello stack, cioè, dall'indirizzo contenuto in SP. Viceversa, il contenuto di queste coppie di registri possono essere immagazzinate nello stesso modo

ad un determinato indirizzo di memoria o sulla cima dello stack. Inoltre, il registro SP può essere caricato da HL, IX ed IY. Ciò facilita la creazione di stack multipli.

La coppia di registri AF può anche essere posta sulla cima dello stack.

La tabella che elenca tutte le possibilità è mostrata nella Figura 4-3. Le operazioni push e pop dello stack sono incluse come parte dei trasferimenti dei dati a 16 bit.

Tutte le operazioni dello stack trasferiscono il contenuto di una coppia di registri a o dallo stack. Notate che non c'è nessuna istruzione push o pop singola per salvare registri ad otto bit individuali.

DESTINAZIONE		SORGENTE: REGISTRO AF	BC	DE	HL	SP	IX	IY	EST. IMM. nn	INDIR. EST. (nn)	REG. INDIR. (SP)
REGISTRO	AF										F1
	BC								01 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
INDIR. EST.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
ISTRUZIONI PUSH → REG. IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

NOTA: Le istruzioni Push e Pop regolano l'SP dopo ogni esecuzione.

ISTRUZIONI POP

Figura 4-3: Istruzioni di caricamento a 16 bit — 'LD', 'PUSH' e 'POP'

Un push o pop di doppio byte è sempre eseguito su una coppia di registri: AF, BC, DE, HL, IX, IY (vedere la fila più in basso e la colonna più a destra nella Figura 4-3). Quando si opera su AF, BC, DE, HL per l'istruzione è richiesto un byte singolo, ottima cosa per l'efficienza. Per esempio, assumiamo che il puntatore dello stack SP contenga il valore "0100". Viene eseguita la seguente istruzione:

PUSH AF

Quando si spinge il contenuto della coppia di registri sullo stack, il puntatore dello stack SP per prima cosa è diminuito, poi il contenuto del registro A è depositato sulla cima dello stack. Poi l'SP è diminuito ancora, e il contenuto di F è depositato sullo stack. Alla fine del trasferimento nello stack, SP punta all'elemento più in alto dello stack, che nel nostro esempio è il valore di F.

È importante ricordare che, nel caso dello Z80, l'SP punta alla *cima* dello stack e l'SP è *diminuito* ogni qual volta è spinta una coppia di registri. In altri processori sono spesso usate altre convenzioni, e questo può essere una fonte di confusione

L'effetto di questa istruzione è illustrato dal seguente schema:

		INDIRIZZAMENTO IMPLICATO				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLICATO	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

Figura 4-4: Istruzioni di scambio 'EX' ed 'EXX'

Istruzioni di scambio

Addizionalmente, per le operazioni di scambio è stato riservato uno mnemonico specializzato EX. EX non è un semplice trasferimento di dati, ma un trasferimento duale di dati. Cambia veramente il contenuto di *due* determinate posizioni. EX può essere usato per scambiare la cima dello stack con HL, IX, IY e anche per scambiare il contenuto di DE, HL, AF ed AF' (ricordatevi che AF' sta per l'altra coppia di registri AF utilizzabile nello Z80).

Infine, è utilizzabile una speciale istruzione EXX per scambiare il contenuto di BC, DE, HL con il contenuto dei registri corrispondenti nel secondo gruppo di registri dello Z80.

Nella figura 4-4 sono riassunti gli scambi possibili.

			SORGENTE	
			REG. INDIR.	
			(HL)	
DESTINAZIONE	REG. INDIR	(DE)	ED A0	'LDI' – Carica (DE) ← (HL) Inc HL & DE, Dec BC
			ED B0	'LDIR', – Carica (DE) ← (HL) Inc HL & DE, Dec BC, ripeti finchè BC = 0
			ED A8	'LDD' Carica (DE) ← (HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Carica (DE) ← (HL) Dec HL & DE, Dec BC, ripeti finché BC = 0

Reg HL punta alla sorgente
Reg DE punta alla destinazione
Reg BC è il contatore dei byte

Figura 4-5: Gruppo di trasferimento di blocco

Istruzioni per il trasferimento di blocchi

Le istruzioni per il trasferimento di blocchi sono istruzioni che generano il trasferimento di un blocco di dati piuttosto che un byte singolo o doppio. Le istruzioni per il trasferimento dei blocchi sono molto complesse da realizzare per il fabbricante e non sono di solito fornite sui microprocessori. Sono utili per la programmazione e possono migliorare le prestazioni di un programma, specialmente durante l'operazione di input output. Attraverso questo libro sarà dimostrato il loro uso ed i vantaggi. Nel caso dello Z80 è disponibile qualche istruzione per il trasferimento automatico di blocchi. Esse usano determinate convenzioni.

Tutte le istruzioni per il trasferimento di blocchi richiedono l'uso di tre coppie di registri: BC, DE, HL:

BC è usato come un contatore a 16 bit. Ciò significa che possono essere mossi automaticamente fino a 2^{16} = 64K byte. HL è usato come puntatore della sorgente. Può puntare dovunque nella memoria. DE è usato come il puntatore della destinazione e può puntare ovunque nella memoria.

Sono fornite quattro istruzioni per il trasferimento di blocchi:

LDD, LDDR, LDI, LDIR

Tutte queste istruzioni decrementano il registro del contatore BC con ogni trasferimento. Due di esse decrementano i registri puntatori DE ed HL, LDD ed LDDR, mentre le altre due incrementano DE ed HL, LDI ed LDIR. Per ognuno di questi due gruppi di istruzioni, la lettera R alla fine dello mnemonico indica una ripetizione automatica. Esaminiamo queste istruzioni.

LDI sta per "caricare e aumentare". Trasferisce un byte dalla posizione di memoria puntata da H ed L alla destinazione nella memoria puntata da D ed E. Diminuisce anche BC. Aumenterà automaticamente H, L e D, E in modo che tutte le coppie di registri sono propriamente condizionate per eseguire il prossimo trasferimento di byte ogni qual volta è richiesto.

LOCAZIONE DI RICERCA

REG. INDIR. (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC Ripeti finchè BC = 0 oppure uguaglianza
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Ripeti finchè BC = 0 oppure uguaglianza

HL punta alla locazione nella memoria che deve essere confrontata con il contenuto dell'accumulatore.
BC è il contatore dei byte.

Figura 4-6: Gruppo di ricerca di blocco

LDIR sta per "caricare aumentare e ripetere", cioè, eseguire LDI ripetutamente fino a che i registri del contatore BC raggiungono il valore di "0". È usato per muovere un blocco continuo di dati automaticamente da un'area di memoria ad un'altra.

LDD e LDDR operano nello stesso modo eccetto che per il fatto che il puntatore d'indirizzi è *diminuito* piuttosto che aumentato. Il trasferimento inizia perciò all'indirizzo *più alto* nel blocco invece del più basso. Il risultato delle quattro istruzioni è riassunto nella Figura 4-5.

Istruzioni automatizzate simili sono disponibili per il CP (confrontare) e sono riassunte nella Figura 4-6.

Istruzioni per l'elaborazione dei dati

Aritmetica

Sono fornite due principali operazioni aritmetiche: l'addizione e la sottrazione. Sono state usate in modo esteso nel precedente capitolo. Ci sono due tipi di addizione, con e senza riporto, rispettivamente ADD e ADC. Allo stesso modo, sono fornite due tipi di sottrazione con o senza riporto. Sono SUB ed SBC.

Addizionalmente, sono fornite tre istruzioni speciali: DAA, CPL e NEG. L'istruzione DAA di regolazione decimale dell'Accumulatore è stata usata per compiere le operazioni BCD. È normalmente usata per ogni addizione o sottrazione BCD. Sono disponibili anche due operazioni

SORGENTE

	INDIRIZZAMENTO DI REGISTRO							REG. INDIR.	INDICIZZATO		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
SOMMA 'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
SOMMA CON RIPORTO 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SOTTRAI 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SOTTRAI CON CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
CONFRONTA 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENTA 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENTA 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Figura 4-7: Operazioni aritmetiche e logiche ad 8 bit

di complementazione. CPL calcolerà il complemento ad uno dell'accumulatore. e NEG negherà l'accumulatore nel suo formato di complemento (complemento a due).

Tutte le precedenti istruzioni operano su dati ad otto bit.

Le operazioni a 16 bit sono più ristrette. ADD, ADC e SBC sono disponibili su registri specifici, come descritto nella Figura 4-8.

Infine, sono disponibili le istruzioni di incremento e decremento le quali operano su tutti i registri, sia nel formato ad otto bit che a 16 bit. Sono elencate nella Figura 4-7 (operazioni ad otto bit) e 4-8 (operazioni a 16 bit).

Notate che, in generale, tutte le operazioni aritmetiche modificano alcuni dei flag. Il loro risultato è ampiamente descritto nell'Appendice alla fine di questo libro. Comunque, è importante notare che le istruzioni INC e DEC che operano sulle coppie di registri non modificano nessuno dei flag. Questo dettaglio è importante da tenere in mente. Ciò significa che se voi aumentate o diminuite una delle coppie di registri al valore "0", il bit Z nel registro dei flag F non sarà posto ad 1. Per far ciò il valore del registro deve essere esplicitamente "provato" col valore "0" nel programma.

È anche importante ricordare che le istruzioni ADC ed SBC interessano sempre tutti i flag. Questo non significa che tutti i flag saranno necessariamente differenti dopo la loro esecuzione. Comunque, potrebbero essere differenti.

DESTINAZIONE		SORGENTE: BC	DE	HL	SP	IX	IY
SOMMA 'ADD'	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
SOMMA CON RIPORTO E POSIZIONA I FLAG 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SOTTRAI CON RIPORTO E POSIZIONA I FLAG 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENTA 'INC'		03	13	23	33	DD 23	FD 23
DECREMENTA 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Figura 4-8: Operazioni aritmetiche e logiche a 16 bit

Operazioni logiche

Sono fornite tre operazioni logiche: AND, OR (inclusivo) e XOR (esclusivo), più una istruzione di paragone CP. Esse operano esclusivamente su dati ad 8 bit. Esaminiamole una alla volta. (Fa parte della Figura 4-7 una tabella che elenca tutte le possibilità ed i codici operativi per queste istruzioni).

AND

Ogni operazione logica è caratterizzata da una *tabella della verità*. che esprime il valore logico del risultato in funzione degli input. La tabella della verità per l'AND è la sequente:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

oppure

AND	0	1
0	0	0
1	0	1

L'operazione AND è caratterizzata dal fatto che l'output è "1" solo se entrambi gli input sono "1". In altre parole, se uno degli input è "0", è garantito che il risultato è "0". Questa caratteristica è usata per azzerare una posizione di bit in una parola. È chiamata "mascheratura".

Uno degli usi importanti dell'istruzione AND è di cancellare o di "mascherare" una o più posizioni di bit determinate in una parola. Presumiamo per esempio di voler azzerare i primi 4 bit a destra in una parola. Questo sarà eseguito dal seguente programma:

```
LD      A,      PAROLA      PAROLA CONTIENE "10101010"
AND             11110000B   "11110000" È LA MASCHERA
```

Presumiamo che sia uguale a "10101010".

Il risultato di questo programma è di lasciare nell'accumulatore il valore "10100000". "B" è usato per indicare un valore binario.

Esercizio 4-1: *Scrivete un programma di tre linee che azzererà i bit 1 e 6 di PAROLA (WORD).*

Esercizio 4-2: *Cosa accade con una MASCHERA* = "1111111"?

OR

Questa istruzione è l'operazione OR inclusivo. È caratterizzata dalla seguente tabella della verità.

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

oppure

OR	0	1
0	0	1
1	1	1

L'OR logico è caratterizzato dal fatto che se uno degli operandi è "1", allora il risultato è sempre "1". L'ovvio uso di OR è quello di mettere ogni bit in una parola ad "1".

Regoliamo i quattro bit più a destra di PAROLA a 1. Il programma è:

```
LD      A,      PAROLA
OR      A,      00001111B
```

Presumiamo che WORD (parola) conteneva "10101010". Il valore finale dell'accumulatore sarà "10101111".

Esercizio 4-3: *Cosa accadrebbe se dovessimo usare l'istruzione OR A, 10101111B?*

Esercizio 4-4: *Qual è il risultato di fare l'OR con "FF" esadecimale?*

XOR

XOR sta per "OR esclusivo". L'OR esclusivo differisce dall'OR inclusivo che abbiamo appena descritto in un aspetto: il risultato è "1" solo se uno, e solo uno, degli operandi è uguale a "1". Se entrambi gli operandi sono uguali ad "1", l'OR normale darebbe un risultato "1". L'OR esclusivo dà un risultato "0". La tabella della verità è:

0 XOR 0 = 0
0 XOR 1 = 1

oppure

1 XOR 0 = 1
1 XOR 1 = 0

XOR	0	1
0	0	1
1	1	0

L'OR esclusivo è usato per paragoni. Se qualche bit è differente, l'OR esclusivo di due parole non sarà zero. Inoltre, nel caso dello Z80, l'OR esclusivo può essere usato per complementare una parola, poiché non c'è nessuna operazione di complemento su qualcosa escluso l'accumulatore. Questo è fatto eseguendo l'XOR di una parola con tutti "1". Il programma compare sotto:

```
LD   A, WORD
XOR  r, 11111111B
```

dove "r" indica il registro.

Presumiamo che WORD contenesse "10101010". Il valore finale del registro sarà "01010101". Potete verificare che questo è il complemento del valore originale.

XOR può essere usato come un "bit toggle" (toggle = ginocchiera).

Esercizio 4-5: *Qual è il risultato di XOR usando un registro con "00" esadecimale?*

Operazioni di skew (Shift e Rotate)

Differenziamo per prima cosa tra le operazioni di shift (spostamento) e di rotate (rotazione) che sono illustrate nella Figura 4-9. In una operazione di shift, il contenuto del registro è spostato a sinistra oppure a destra di una posizione di bit. Il bit che cade fuori dal registro entra nel bit di riporto C, e il bit che entra è zero. Questo è stato spiegato al paragrafo precedente.

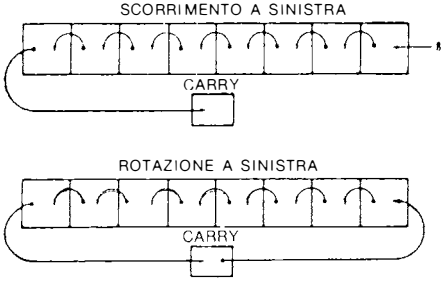


Figura 4-9: Scorrimento e Rotazione

Esiste una eccezione: è l'*aritmetic-shift-right*: spostamento a destra aritmetico.

Quando si eseguono operazioni su numeri negativi nel formato in complemento a due, il bit più a sinistra è il bit del segno. Nel caso di numeri negativi è "1". Quando si divide un numero negativo per "2" spostandolo alla destra, dovrebbe rimanere negativo, cioè, il bit più a sinistra

dovrebbe rimanere "1". Questo è eseguito automaticamente dall'istruzione SRA o Shift Right Aritmetic (spostamento a destra aritmetico).

In questo "aritmetic shift right", il bit che entra a sinistra è identico al bit del segno. È "0" se il bit più a sinistra era "0", e "1" se il bit più a sinistra era "1". Ciò è illustrato alla destra della Figura 4-10, la quale mostra tutte le possibili operazioni di shift e di rotate.

Rotazioni

Una rotazione differisce da uno spostamento per il fatto che il bit che entra nel registro è quello che cadrà dall'altra estremità del registro o dal bit di riporto. Nel caso dello Z80 sono forniti due tipi di rotazioni: una rotazione ad otto bit e una rotazione a nove bit.

La rotazione a nove bit è illustrata nella Figura 4-11. Per esempio, nel caso di una rotazione a destra, gli otto bit del registro sono spostati a destra di una posizione di bit. Il bit che cade fuori dalla parte destra del registro va, come di solito, nel bit del riporto. A questo punto il bit che entra all'estremità sinistra del registro è il valore precedente del bit del riporto (prima che sia riscritto con il bit che cade). In matematica questa è chiamata la rotazione a nove bit poiché gli otto bit del registro più il nono bit (il bit di riporto) sono ruotati a destra di una posizione di bit. Viceversa, la rotazione a sinistra realizza lo stesso risultato nella direzione opposta.

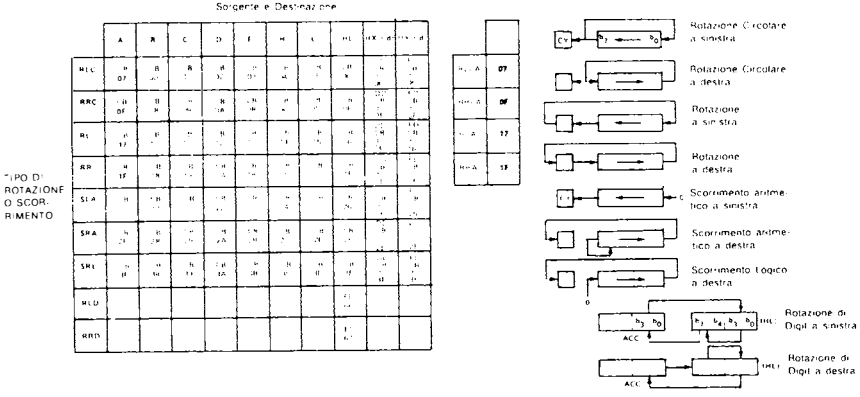


Figura 4-10: Rotazione e Scorrimento

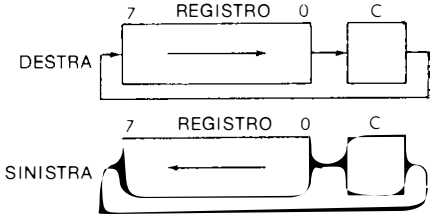


Figura 4-11: Rotazione a 9 bit

La rotazione ad otto bit opera in un modo simile. Il bit "0" è copiato nel bit "7", oppure il bit sette è copiato nel bit 0, a secondo della direzione della rotazione. Inoltre, anche il bit che esce dal registro è copiato nel bit del riporto. Ciò è illustrato dalla Figura 4-12.

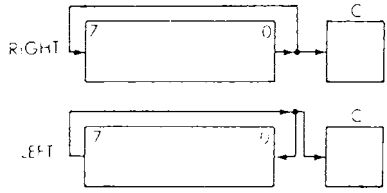


Figura 4-12: Rotazione ad 8 bit

Speciali istruzioni di Digit

Per facilitare l'aritmetica BCD sono fornite due speciali istruzioni di rotazione di digit. Il risultato è una rotazione a 4 bit tra due digit contenuti nella locazione di memoria puntata tramite i registri HL e un digit nella parte più bassa dell'accumulatore. Questo è illustrato dalla Figura 4-13.

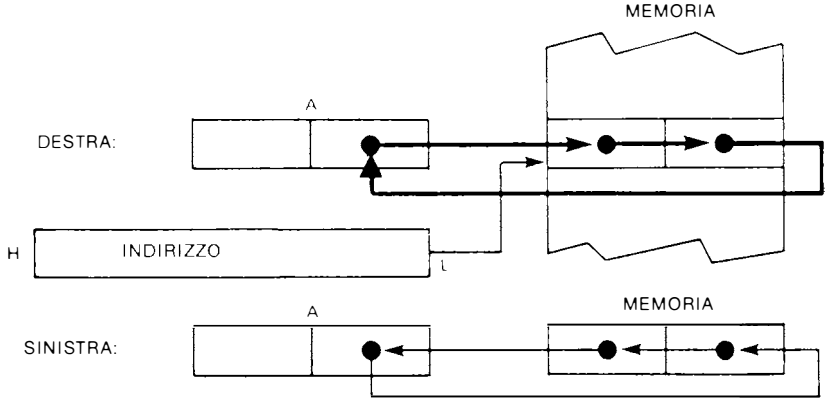


Figura 4-13: Istruzioni di rotazione di digit

Manipolazione dei bit

Sopra è stato mostrato come possono essere usate le operazioni logiche per il set o per il reset di bit o gruppi di bit in determinati registri. Comunque, è conveniente poter eseguire il set o reset di qualsiasi registro o locazione di memoria con un'istruzione singola. Questa capacità richiede un considerevole numero di codici operativi e perciò di solito non è fornita sulla maggior parte dei microprocessori.

Comunque, lo Z80 è fornito di estensive capacità per la manipolazione dei bit.

Esse sono mostrate nella Figura 4-14. Questa tabella include anche le istruzioni di prova che saranno descritte nel prossimo paragrafo. Sono anche disponibili due speciali istruzioni per operare sul flag di riporto. Sono il CCF (Complement Carry Flag) e l'SCF (Set Carry Flag). Sono mostrati nella Figura 4-15.

Test e Jump

Poiché le operazioni di prova fanno molto affidamento sull'uso del registro dei flag, qui noi descriveremo in dettaglio, il ruolo di ognuno dei flag. Il contenuto del registro dei flag compare nella Figura 4-16.

	BIT	NOMI RIZZAMENTO DI REGISTRO							REG INDIR	INDICIZZATO	
		A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Figura 4-14: Gruppo per la manipolazione di bit

Aggiustamento Decimale Accumulatore 'DAA'	27
Complementa Accumulatore 'CPL'	2F
Cambia segno Accum. 'NEG' (Complemento a 2)	ED 44
Complementa il Flag di Riporto 'CCF'	3F
Poni ad 1 il Flag di Riporto 'SCF'	37

Figura 4-15: Operazioni per scopi generali sui registri AF

7	6	5	4	3	2	1	0
S	Z		H		P/V	N	C
(T)	(T)				(T)		(T)

Figura 4-16: Il registro dei flag

C è il riporto, N è addizione o sottrazione P/V è parità o overflow, H è mezzo riporto, Z è zero, S è il segno. Il bit 3 e 5 del registro dei flag non sono usati ("0"). I due flag H e N sono usati per l'aritmetica BCD e non possono essere provati. Gli altri quattro flag (C, P V, Z, S) possono essere provati in corrispondenza alle istruzioni di jump o di chiamata condizionale.

Ora sarà descritto il ruolo di ogni flag.

Riporto (C)

Nel caso di quasi tutti i microprocessori, e in particolare dello Z80, il bit del riporto assume un ruolo duale. Per primo, è usato per indicare se un riporto (o prestito) è risultato in una operazione di addizione o sottrazione.

Secondo, è usato come un nono bit nel caso di operazioni di shift e rotate. Usando un bit singolo per eseguire entrambi i ruoli facilita alcune operazioni, come l'operazione di moltiplicazione. Questo dovrebbe essere chiaro dalla spiegazione della moltiplicazione che è stata presentata nel capitolo precedente.

Quando imparate ad usare il bit del carry (riporto), è importante ricordare che tutte le operazioni aritmetiche influenzano a seconda del risultato delle istruzioni.

Allo stesso modo, tutte le operazioni di shift e di rotate usano il bit del carry e lo influenzano a seconda del valore del bit che esce dal registro.

Nel caso delle istruzioni logiche (AND, OR, XOR), il bit del carry sarà sempre resettato. Possono essere usate per azzerare il carry (riporto) in modo esplicito.

Le istruzioni che interessano il bit del carry sono: ADD A, s; ADC A, s; SUB s; SBC A, s; CP s; NEG; AND s; OR s; XOR s; ADD DD, ss; ADC HL, ss; SBC HL, ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; DDA; SCF; CCF; NEG s.

Sottrazione (N)

Questo flag non è normalmente usato dal programmatore, ed è usato dallo stesso Z80 durante le operazioni BCD.

Il lettore ricorderà dal precedente capitolo che, dopo una addizione o sottrazione BCD, è eseguita una istruzione DAA (Decimal Adjust Accumulator) per ottenere risultati BCD validi. Comunque, l'operazione di "regolazione" DAA è differente dopo una addizione e dopo una sottrazione. Perciò, il DAA agisce in modo differente a seconda del valore del flag N. Il flag N è regolato a "0" dopo una addizione ed è regolato a "1" dopo una sottrazione.

Il simbolo usato per questo flag, "N", può fare confusione al programmatore che ha usato altri processori, poiché può essere confuso per il bit di segno. È un bit di segno per un'operazione interna. N è posto a "0" da: ADD A, s; ADC A, s; AND, s; OR, s; XOR, s; INC, s; ADD DD, ss; ADC HL, ss; RLA; RLCA; RRA; RRCA; RL, m; RLC, m; RR, m; RRC, m; SLA, m; SRA, m; SRL, m; RLD; RRD; SCF; CCF; IN r, (C); LDI; LDD; LDIR; LDDR; LD A, I; LD A, r; BIT b, s.

N è posto a "1" da: SUB s; SBC A, s; CP s; NEG; DEC m; SBC HL, ss; CPL; INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR.

Parità/Overflow (P/V)

Il flag della parità/overflow compie due diverse funzioni. Determinate istruzioni influenzeranno questo flag a seconda della parità del risultato; la parità è determinata contando il numero totale di uni nel risultato. Se questo numero è dispari, il bit di parità sarà regolato a "0" (parità dispari); se è pari, sarà regolato a "1" (parità pari). La parità è più frequentemente usata sui blocchi di caratteri (di solito nel formato ASCII). Il bit della parità è un bit addizionale che è aggiunto al codice a sette bit che rappresenta il carattere, per verificare l'integrità di dati che sono stati immagazzinati in un dispositivo di memoria. Per esempio, se un bit nel codice che rappresenta il carattere è stato cambiato accidentalmente, a causa di un malfunzionamento nel dispositivo di memoria (come una memoria a disco o RAM), o durante la trasmissione, allora sarà stato cambiato il numero totale di uni nel codice a sette bit. Controllando il bit della parità, sarà scoperta la differenza e sarà evidenziato l'errore. In particolare, il flag è usato con istruzioni logiche e di rotazione. Poi, naturalmente, durante una operazione di input da un dispositivo I/O, il flag della parità indicherà la parità dei dati che sono letti.

Per il lettore che conosce l'Intel 8080, notate che il flag della parità nell'8080 è usato esclusivamente come tale.

Nel caso dello Z80, é usato per diverse funzioni addizionali. Perciò, questo flag dovrebbe essere trattato con attenzione quando si passa da un microprocessore all'altro.

Nel caso dello Z80, il secondo essenziale uso di questo flag è come flag di overflow (non disponibile nell'8080). Il flag di overflow è stato descritto nel Capitolo 1, quando è stata introdotta la numerazione in complemento a due. Tale flag rivela il fatto che, durante una addizione o sottrazione, il segno del risultato è "accidentalmente" cambiato a causa dell'overflow del risultato nel bit di segno. (Ricordatevi che, usando una rappresentazione ad otto bit, il numero positivo più grande è +127, ed il numero negativo più piccolo è −128 in complemento a due). Infine, nel caso dello Z80, questo bit è usato per due funzioni senza legami tra loro.

Durante il trasferimento di blocchi di istruzioni (LDD, LDDR, LDI, LDIR), e durante le istruzioni di ricerca (CPD, CPDR, CPI, CPIR), questo flag è usato per scoprire se il registro B del contatore ha raggiunto il valore "0". Con istruzioni di decremento questo flag è resettato a "0" se è "0" la coppia dei registri contenenti il contatore di byte. Con quelle del tipo "ad aumento" è posto ad 1 se BC−1 = 0 all'inizio delle istruzioni, cioé, se BC sarà diminuito a "0" dell'istruzione.

Infine, quando si eseguono le due istruzioni speciali LD A, I e LD A, R, il flag P/V riflette il valore del flip-flop che abilita l'interrupt (IFF2). Questa caratteristica può essere usata per conservare o provare questo valore.

Il flag P è influenzato da: AND, s; OR s; XOR s; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r, (C).

Il flag V è influenzato da: ADD A, s; ADC A, s; SUB s; SBC A, s; CP s; NEG; INC s; DEC m; ADC HL, ss; SBC HL, ss; NEG.

È anche usato da: LDIR; LDDR (posto a "0"); LDI; LDD; CPI; CPIR; CPD; CPDR.

Il flag del mezzo riporto (H)

Il flag del mezzo riporto indica un eventuale riporto dal bit 3 al bit 4 durante una operazione

aritmetica. In altre parole, rappresenta il riporto dal nibble di ordine inferiore (gruppo di 4 bit) in quello di ordine superiore. Chiaramente, è principalmente usato per le operazioni BCD. In particolare, è usato internamente al microprocessore dall'istruzione Decimal Adjust Accumulator in modo da regolare il risultato al suo valore corretto.

Il flag sarà posto ad 1 durante un'addizione quando c'è un riporto dal bit 3 al bit 4 e azzerato quando non c'è nessun riporto. Viceversa, durante un'operazione di sottrazione, sarà posto ad 1 se c'è un prestito dal bit 4 al bit 4, e azzerato se non c'è nessun prestito.

Il flag sarà condizionato dall'operazione di addizione, sottrazione, incremento, decremento, paragone e dalle operazioni logiche.

Le istruzioni che influenzano il bit H sono: ADD A, r; ADD A, s; SUB s; SBC A. s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SR m; SRL m; RLD; RRD; DAA; CPL; SCF, IN r, (C); LDI; LLD; LDIR; LDDR; LD A; LD A, r; BIT b, r; NEG s.

Notate che il bit H non è influenzato dalle istruzioni di addizione e sottrazione a 16 bit.

Zero (Z)

Il flag è usato per indicare se il valore di un byte che è stato calcolato, e che sta per essere trasferito, è zero. È anche usato per le istruzioni di confronto per indicare uguaglianza e per altre funzioni miste.

Nel caso di un'operazione che termina con risultato zero, o di un trasferimento di dati, il bit Z è posto ad 1 ogni volta che il byte è zero. Altrimenti Z è resettato a "0".

Nel caso delle istruzioni di confronto, il bit Z è posto ad "1" ogni volta che il confronto riesce e, nel caso contrario, è posto a "0".

Inoltre, nel caso dello Z80, è usato per tre altre funzioni: è usato con l'istruzione BIT per indicare il valore di un bit che sta per essere provato. È posto ad "1" se il bit determinato è "0" e resettato nel caso contrario.

Con le speciali istruzioni di input/output di blocco (INI, IND, OUTI, OUTD), il flag Z è posto ad 1 se $D-1=0$, e resettato in caso contrario; è posto ad 1 se il contatore dei byte sarà diminuito a "0" (INIR, INDR, OTIR, OTDR).

Infine, con le istruzioni speciali IN r, (C), il flag Z è posto ad "1" per indicare che il byte d'input ha il valore "0".

In sommario, le seguenti istruzioni condizionano il valore del bit Z: ADD A, s; ADC A, s; SUB s; SBC A, s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL, ss; SBC HL, ss; RL m; RCL m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r (C); INI: IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR, CPI; CPIR; CPD; CPDR; LD A, I; LD A, r; BIT b, s; NEG s.

Le istruzioni abituali che non influenzano il bit Z sono: ADD DD, ss; RLA; RLCA; RRA; RRCA; CPL; SCF; CCF; LDI; LDD; LDIR; LDDR; INC DD; DEC DD.

Segno (S)

Questo flag riflette il valore del bit più significativo di un risultato oppure di un byte che sta per essere trasferito (bit sette). Nella numerazione in complemento a due, il bit più significativo è usato per rappresentare il segno. "0" indica un numero positivo ed "1" indica un numero negativo. Quindi il bit sette è chiamato il bit di segno.

Nel caso della maggior parte dei microprocessori, il bit di segno gioca un ruolo importante quando comunica con i dispositivi input/output. La maggiore parte dei microprocessori non sono forniti di una istruzione BIT per provare il contenuto di qualsiasi bit in un registro o nella memoria.

Come risultato, il bit di segno, è di solito il bit più conveniente da provare. Quando si esamina lo stato di un dispositivo input/output, il leggere il registro di stato condizionerà automaticamente il bit di segno, il quale sarà posto uguale al valore del bit sette del registro di stato. Può allora essere convenientemente provato dal programma. Ecco perché il registro di stato della maggior parte dei chip di input/output collegati ai sistemi a microprocessore hanno il loro indicatore più importante (di solito indicatore di "pronto/non pronto") nella posizione di bit sette.

Nel caso dello Z80 è fornita una speciale istruzione BIT. Comunque, per provare una posi-

zione di memoria (che può essere l'indirizzo di un registro di stato d'input/output), l'indirizzo per prima cosa deve essere caricato nei registri IX, IY oppure HL. Non è fornita nessuna istruzione bit per provare direttamente un determinato indirizzo della memoria (cioè, per questa istruzione non c'è nessun modo di indirizzamento diretto). L'importanza di mettere il flag di pronto/non pronto, in un dispositivo I/O, nel bit sette, rimane comunque intatto, anche nel caso dello Z80.

Infine, il flag di segno è usato dalla speciale istruzione IN, (C) per indicare il segno dei dati che si stanno leggendo.

Le istruzioni che influenzano il bit di segno sono: ADD A, s; SUB s; SBC A, s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL, ss; SBC HL, ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r, (C); CPR; CPIR; CPD; CPDR; LD A, I; LD A, r; NEG.

Sommario sui flag

I bit di flag sono usati per rivelare automaticamente condizioni speciali all'interno dell'ALU del microprocessore. Possono essere convenientemente provati tramite istruzioni specializzate, in modo che l'azione specifica può essere presa in risposta alla condizione rivelata. È importante capire il ruolo dei vari indicatori utilizzabili, poiché la maggior parte delle decisioni prese entro il programma saranno prese in funzione di questi bit di flag. Tutti i jump (salti) eseguiti all'interno di un programma salteranno a determinate posizioni a seconda dello stato di questi flag. La sola eccezione implica il meccanismo di interrupt che sarà descritto nel capitolo sull'input/output e può causare un salto a determinate posizioni ogni qual volta è ricevuto un segnale hardware su pin (piedini) specializzati dello Z80.

A questo punto, è necessario ricordare soltanto la funzione principale di ognuno di questi bit. Quando si programma, il lettore può riferirisi alla descrizione delle istruzioni nella sezione dell'Appendice del libro per verificare l'effetto di ogni istruzione dei vari flag. La maggior parte dei flag può essere ignorata il più delle volte, e il lettore che non è ancora familiare con essi non dovrebbe sentirsi intimidito dalla loro apparente complessità. Il loro uso diventerà più chiaro quando esamineremo altri programmi applicativi. Nella Figura 4-17 è mostrato un sommario dei sei flag e il modo in cui sono condizionati dalle varie istruzioni.

Le istruzioni di jump

Una istruzione branch (ramificazione) è un'istruzione che provoca una ramificazione forzata ad un determinato indirizzo del programma. Cambia il normale flusso di esecuzione del programma da un modo sequenziale in uno dove è improvvisamente eseguito un segmento differente del programma. I jump possono essere condizionati o incondizionati. Un jump incondizionato è quello in cui avviene la ramificazione ad un determinato indirizzo, senza curarsi di ogni altra condizione.

Un salto condizionato è quello che avviene ad un determinato indirizzo solo se sono soddisfatte una o più condizioni. Questo è il tipo di istruzione jump usata per prendere decisioni basate sui dati o risultati calcolati.

Per spiegare le istruzioni di jump condizionato, è necessario capire il ruolo del registro dei flag, poiché tutte le decisioni di ramificazione sono basate su questi flag. Questo era lo scopo del paragrafo precedente. Qui ora possiamo esaminare più dettagliatamente le istruzioni di jump fornite dallo Z80.

Sono forniti due principali tipi di istruzioni jump: le istruzioni jump all'interno del programma principale e il tipo speciale di istruzioni di branch (ramificazione) usato per saltare ad una subroutine e per ritornare da essa ("call" e "return"). Come risultato di ogni istruzione jump, il contatore di programma PC sarà ricaricato con un nuovo indirizzo e la solita esecuzione del programma riprenderà da questo punto in avanti. La potenza totale delle varie istruzioni jump può essere compresa solo nel contesto dei vari modi di indirizzamento forniti dal microprocessore. Questa parte della trattazione sarà rimandata al prossimo capitolo, dove sono discussi i modi di indirizzamento. Qui, considereremo soltanto gli altri aspetti di queste istruzioni.

ISTRUZIONE	C	Z	P/V	S	N	H	COMMENTI
ADD A, s; ADC A, s	↕	↕	V	↕	0	↕	Somma ad 8 bit o somma con riporto
SUB s; SBC A, s, CP s, NEG	↕	↕	V	↕	1	↕	Sottrazione ad 8 bit, sottrazione con riporto, confronta e cambia segno al contenuto dell'accumulatore
AND s	0	↕	P	↕	0	1	Operazioni logiche
OR s; XOR s	0	↕	P	↕	0	0	Poni ad 1 i flag diversi
INC s	•	↕	V	↕	0	↕	Incrementa ad 8 bit
DEC m	•	↕	V	↕	1	↕	Decrementa ad 8 bit
ADD DD, ss	↕	•	•	•	0	X	Somma a 16 bit
ADC HL, ss	↕	↕	V	↕	0	X	Somma a 16 bit con riporto
SBC HL, ss	↕	↕	V	↕	1	X	Sottrazione a 16 bit con riporto
RLA; RLCA, RRA, RRCA	↕	•	•	•	0	0	Ruota l'accumulatore
RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	↕	↕	P	↕	0	0	Ruota e sposta la locazione m.
RLD, RRD	•	↕	P	↕	0	0	Ruotare il digit a sinistra e a destra
DAA	↕	↕	P	↕	•	↕	Aggiusta Decimali accumulatore
CPL	•	•	•	•	1	1	Complementa l'accumulatore
SCF	1	•	•	•	0	0	Poni ad 1 il riporto
CCF	↕	•	•	•	0	X	Complementa il riporto
IN r, (C)	•	↕	P	↕	0	0	Registro d'input indiretto
INI; IND; OUTI; OUTD	•	↕	X	X	1	X	Input ed output dei blocchi
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0 se B ≠ 0 altrimenti Z = 1
LDI, LDD	•	X	↕	X	0	0	Istruzioni di trasferimento di blocco
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 se BC ≠ 0, altrimenti P/V = 0
CPI, CPIR, CPD, CPDR	•	↕	↕	↕	1	X	Istruzioni di ricerca di blocco Z = 1 se A = (HL), altrimenti Z = 0 P/V = 1 se BC ≠ 0, altrimenti P/V = 0
LD A, I; LD A, R	•	↕	IFF	↕	0	0	Il contenuto del flip-flop che abilita l'interrupt (IFF) è copiato nel flag P/V
BIT b, s	•	↕	X	X	0	1	Il complemento del bit b della locazione è copiato nel flag Z
NEG	↕	↕	V	↕	1	↕	Cambia segno all'accumulatore

In questa tabella sono utilizzate le seguenti notazioni:

SIMBOLO	OPERAZIONE
C	Flag di Collegamento/Riporto. C = 1 se l'operazione ha prodotto un riporto dell'MSB dell'operando o del risultato
Z	Flag Zero. Z = 1 se il risultato dell'operazione è zero.
S	Flag segno. S = 1 se l'MSB del risultato è uno.
P/V	Flag di parità o di eccedenza di capacità (overflow). La parità (P) e l'overflow (V) condividono lo stesso flag. Le operazioni logiche influenzano questo flag con la parità del risultato mentre le operazioni aritmetiche influenzano questo flag con l'eccedenza di capacità del risultato. Se P/V contiene la parità, P/V = 1 se il risultato dell'operazione è pari, P/V = 0 se il risultato è dispari. Se P/V contiene l'overflow, P/V = 1 se il risultato dell'operazione ha prodotto un overflow.
H	Flag di mezzo riporto. H = 1 se l'operazione di somma o di sottrazione ha prodotto un riporto o chiesto in prestito dal bit 4 dell'accumulatore.
N	Flag di addizione/sottrazione. N = 1 se la precedente operazione era una sottrazione.
	I flag H e N sono usati in congiunzione con l'istruzione di aggiustamento decimale (DAA) per correggere propriamente il risultato nel formato BCD compattato che segue l'addizione o la sottrazione usando operandi con formati BCD compattati.
	Il flag è influenzato a seconda del risultato dell'operazione.
•	Il flag non è variato dall'operazione.
0	Il flag è resettato (a zero) dall'operazione.
1	Il flag è posto (ad uno) dall'operazione.
X	Il flag è non significativo.
V	Il flag P/V è influenzato a seconda del risultato di overflow dell'operazione.
P	Il flag P/V è influenzato a secondo del risultato di parità dell'operazione
r	Ognuno dei registri della CPU A, B, C, D, E, H, L.
s	Qualsiasi locazione ad 8 bit per tutti i modi di indirizzamento permessi per l'istruzione particolare
ss	Qualsiasi locazione a 16 bit per tutti i modi di indirizzamento permessi per quella istruzione.
ii	Ognuno dei due registri indice IX od YX.
R	Contatore di rinfresco.
n	Valore ad 8 bit nel range < 0, 255 > .
nn	Valore a 16 bit nel range < 0, 65535 > .
m	Qualsiasi locazione ad 8 bit per tutti i modi di indirizzamento permessi per la particolare istruzione.

Figura 4-17: Sommario delle operazioni sui flag

I jump possono essere incondizionati (che si ramificano ad un determinato indirizzo della memoria) oppure condizionati. Nel caso di un jump condizionato, può essere provato uno dei quattro bit di flag. Sono i flag Z, C, P/V ed S. Ognuno di essi può essere provato per il valore "0" oppure "1".

Le abbreviazioni corrispondenti sono:

Z =ZERO (Z=0)
NZ =non zero (Z=1)
C =riporto (C=1)
NC =nessun riporto (0=C)
PO =parità dispari
PE =parità pari
P =positivo (S=0)
M =meno (S=1)

In aggiunta, nello Z80 è utilizzabile una speciale istruzione di combinazione che diminuirà il registro B ed eseguirà il jump ad un determinato indirizzo della memoria fino a quando non è zero. Questa è una potente istruzione usata per terminare un loop, ed è già stata usata diverse volte nel capitolo precedente: è l'istruzione DJNZ.

Allo stesso modo, le istruzioni CALL e RET (ritorno) possono essere condizionate o incondizionate. Provano gli stessi flag come l'istruzione branch (ramificazione) che abbiamo già descritto.

L'utilizzabilità di ramificazioni condizionali è una risorsa potente in un computer e non è generalmente fornita sugli altri microprocessori ad otto bit. Migliora l'efficienza dei programmi compiendo in una istruzione singola quello che altrimenti richiederebbe due istruzioni. Infine, nel caso di routine d'interrupt sono state fornite due speciali istruzioni di ritorno. Sono RETI e RETN. Saranno descritte nel paragrafo sugli interrupt del Capitolo 6.

I modi di indirizzamento e i codici operativi per le varie ramificazioni utilizzabili sono mostrate nella Figura 4-18.

			CONDIZIONE									
			NON COND	CARRY	NON CARRY	ZERO	NON ZERO	PARITA PARI	PARITA DISPARI	SEGNO NEG	SEGNO POS	REG B=0
JUMP 'JP'	IMMED EST	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVO	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG INDIR	(HL)	E9									
JUMP 'JP'	REG INDIR	(IX)	DD E9									
JUMP 'JP'	REG INDIR	(IY)	FD E9									
'CALL'	IMMED EST	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENTA B. SALTA SE NON ZERO 'DJNZ'	RELATIVO	PC+e										10 e-2
RITORNO 'RET'	REG INDIR	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RITORNO DA INT 'RETI'	REG INDIR	(SP) (SP+1)	ED 4D									
RITORNO DA INT NON MASCHERABILE 'RETN'	REG INDIR	(SP) (SP+1)	ED 45									

Figura 4-18: Istruzioni di jump

Nel Capitolo 5 è presentata una trattazione dettagliata dei vari modi di indirizzamento.

Esaminando la Figura 4 18 diventa evidente che molti modi di indirizzamento sono limitati. Per esempio, il jump assoluto JP nn può provare quattro flag, mentre JR può solo provare due flag.

Notate una importante osservazione. JR tende ad essere usato ogni qual volta è possibile siccome è più corto di JP (un byte meno) e facilita il riposizionamento del programma.

Comunque, JR e IP non sono intercambiabili; JR non può provare la parità oppure i flag di segno.

È utilizzabile un altro tipo di branch specializzato; questa è l'istruzione *restart* (rimessa in marcia) o RST. È una istruzione di un byte che permette il salto ad ognuno degli otto indirizzi di avviamento all'estremità bassa della memoria.

I suoi indirizzi di avviamento sono, nel decimale, 0, 8, 16, 24, 32, 40, 48 e 56. È una istruzione potente perché è resa effettiva in un singolo byte.

È la ramificazione più veloce che possa essere usata, e per questa ragione, è essenzialmente usata per rispondere agli interrupt.

Comunque, è anche utilizzabile dal programmatore per altri usi.

Nella Figura 4-13 è mostrato un riassunto dei codici operativi per questa istruzione.

INDIRIZZO DI CHIAMATA	OP CODE	
0000_H	C7	'RST 0'
0008_H	CF	'RST 8'
0010_H	D7	'RST 16'
0018_H	DF	'RST 24'
0020_H	E7	'RST 32'
0028_H	EF	'RST 40'
0030_H	F7	'RST 48'
0038_H	FF	'RST 56'

Figura 4-19: Gruppo di Restart

Istruzioni input/output

Le tecniche di input/output saranno descritte dettagliatamente nel Capitolo 6. Semplicemente, i dispositivi input/output possono essere indirizzati in due modi: come posizioni di memoria, usando qualsiasi delle istruzioni che sono già state descritte, o usando determinate istruzioni input/output. Le solite istruzioni ad indirizzamento della memoria usano tre byte: un byte per il codice operativo e due byte per l'indirizzo. Come risultato, sono lente da eseguire, poichè richiedono tre accessi alla memoria. Lo scopo principale delle istruzioni input output specializzate è quello di fornire istruzioni più corte e perciò più veloci. Comunque, le istruzioni input/output hanno due svantaggi.

Per prima cosa, "sprecano" diversi dei pochi e preziosi codici operativi disponibili (poiché di solito sono usati solo 8 bit per fornire tutti i codici operativi necessari per un microprocessore).

Secondo, richiedono la generazione di uno o più segnali input output specializzati, e quindi "sprecano" uno o più dei pochi pin utilizzabili nel microprocessore. Il numero di pin è di solito limitato a 40. A causa di questi possibili svantaggi, determinate istruzioni input/output non sono fornite nell'8080 originale (il primo potente microprocessore general-pourpose ad otto bit introdotto) e nello Z80, che noi sappiamo che è compatibile con l'8080.

Il vantaggio delle istruzioni input/output è quello di essere eseguite più velocemente richiedendo solo due byte. Comunque, un risultato simile può essere ottenuto fornendo uno speciale modo di indirizzamento chiamato indirizzamento della "pagina 0", dove l'indirizzo è limitato ad un campo di otto bit. Questa soluzione è spesso scelta in altri microprocessori.

Le due istruzioni input/output fondamentali sono IN e OUT. Trasferiscono il contenuto delle posizioni I/O specificate in qualsiasi dei registri funzionanti oppure il contenuto del registro nel dispositivo I/O. Naturalmente, sono lunghe due byte.

Il primo byte è riservato per il codice operativo (opcode), il secondo byte dell istruzione forma la parte bassa dell'indirizzo. L'accumulatore è usato per fornire la parte più alta dell'indirizzo. È perciò possibile scegliere un dispositivo su 64 K. Comunque, ciò richiede che l'accumulatore sia caricato ogni volta con l'appropriato contenuto, e questo potrebbe rallentare l'esecuzione.

Nel modo a *"register-interrupt"*, il cui formato è IN r, (C), la coppia di registro B e C è usata come puntatore al dispositivo I/O. Il contenuto di B è posto nella parte alta del bus degli indirizzi. Il contenuto del dispositivo I/O specificato è poi caricato nel registro designato da r.

In più, lo Z80 fornisce un modo a registro indiretto più quattro istruzioni specializzate per il trasferimento di blocchi per l'input e l'output.

Lo stesso si applica all'istruzione OUT. Le quattro istruzioni per il trasferimento di blocchi sull'input sono: INI, INIR (INI ripetuto), IND e INDR (IND ripetuto). Allo stesso modo, sull'output, sono: OUTI, OUTIR, OUTD, OUTDR.

In questo trasferimento di blocchi automatizzato, la coppia di registri H ed L è usata come puntatore della destinazione. Il registro C è usato come selettore di dispositivi I/O (uno su 256 dispositivi). Nel caso dell'istruzione output, H ed L puntano alla destinazione Il registro B è usato come contatore e può essere aumentato o diminuito.

Le istruzioni corrispondenti sull'input sono INI quando si incrementa e IND quando si diminuisce.

INI è un trasferimento di un byte singolo automatizzato.

Il registro C sceglie il dispositivo input. Dal dispositivo è letto un byte ed è trasferito all'indirizzo di memoria puntato da H ed L. H ed L vengono quindi aumentati di uno, e il contatore B è diminuito di 1.

INIR è la stessa istruzione, automatica. È eseguita ripetutamente fino a che il contatore diminuisce a "0". Così, può essere trasferita automaticamente fino a 256 byte. Notate che per raggiungere un trasferimento totale di esattamente 256, il registro B dovrebbe essere regolato al valore "0" prima di eseguire questa istruzione. Nella Figura 4-20 e 4-21 sono riassunti i codici operativi per le istruzioni input ed output.

Istruzioni di controllo

Le istruzioni di controllo sono istruzioni che modificano il modo di funzionamento della CPU o manipolano le sue informazioni di stato interno. Sono fornite sette di queste istruzioni.

L'istruzione NOP è una istruzione di "non operazione" la quale non fa niente per un ciclo. È tipicamente usato per introdurre un ritardo deliberato (4 stati = 2 microsecondi con un clock di 2 MHz), oppure per riempire gli spazi creati in un programma durante la fase di messa a punto (debugging). Per facilitare la messa a punto del programma, il codice operativo per il NOP è tradizionalmente tutti zeri. Questo perche, al momento dell'esecuzione, la memoria è spesso cancellata cioè, tutti zeri. L'esecuzione dei NOP è garantito che non provoca nessun danno e non arresta l'esecuzione del programma.

L'istruzione HALT è usata in congiunzione con gli interrupt o un reset". Sospende veramente il funzionamento della CPU. La CPU poi ricomincerà a funzionare ogni qual volta è ricevuto

un segnale di interrupt o di reset. In questo modo, la CPU fa eseguire il NOP. Un halt è spesso posto alla fine dei programmi durante la fase di messa a punto, siccome non c'è niente altro che deve essere fatto dal programma principale. Poi, questo programma deve essere esplicitamente messo in moto.

Sono usate due istruzioni specializzate per impedire, disabilitare o abilitare il flag interno d'interrupt. Esse sono EI e DI. Gli interrupt saranno descritti nel Capitolo 6.

			SORGENTE							
			REGISTRO							REG. IND.
			A	B	C	D	E	H	L	(HL)
'OUT'	IMMED.	(n)								
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' — OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTDR' — OUTPUT Dec HL & B, RIPETI SE B # 0	REG. IND.	(C)								ED B3
'OUTD' — OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTIR' — OUTPUT Inc HL, Dec B, RIPETI SE B # 0	REG. IND.	(C)								ED BB

COMANDI DI OUTPUT DI BLOCCO

INDIRIZZO DELLA PORTA DI DESTINAZIONE

Figura 4-20: Gruppo delle istruzioni di Output

DESTINAZIONE INPUT			INDIRIZZO DELLA PORTA SORGENTE	
			IMMED	REG. INDIR
			(n)	(C)
INPUT 'IN'	INDIRIZZAMENTO DI REG	A	DB n	ED 78
		B		ED 40
		C		ED 48
		D		ED 50
		E		ED 58
		H		ED 60
		L		ED 68
'INI' — INPUT & Inc HL, Dec B	REG. INDIR	(HL)		ED A2
'INIR' — INP, Inc HL, Dec B, RIPETI SE B≠0				ED B2
'IND' — INPUT & Dec HL, Dec B				ED AA
'INDR' — INPUT Dec HL, Dec B, RIPETI SE B≠0				ED BA

COMANDI DI IMPUT DI BLOCCO

Figura 4-21: Gruppo delle istruzioni di Input

Il flag di interrupt è usato per autorizzare o non autorizzare l'interruzione di un programma. Per impedire che l'interrupt avvenga durante qualsiasi determinata porzione di un programma, il flip-flop di interrupt (flag) può essere disabilitato da questa istruzione. Sarà usata nel capitolo 6. Queste istruzioni sono mostrate nella Figura 4-22.

'NOP'	00	
'HALT'	76	
DISABILITA INT '(DI)'	F3	
ABILITA INT '(EI)'	FB	
PONI NEL MODO 0 'IM0'	ED 46	MODO 8080A
PONI NEL MODO 1 'IM1'	ED 56	CHIAMATA ALLA LOCAZIONE 0038_H
PONI NEL MODO 2 'IM2'	ED 5E	CHIAMATA INDIRETTA USANDO IL REGISTRO I E 8 BIT DEL DISPOSITIVO D'INTERRUZIONE COME UN PUNTATORE

Figura 4-22: Esempi di istruzioni ai controllo della CPU

Infine, nello Z80 sono forniti tre modi di interrupt. (Solo uno è disponibile nell'8080). Il modo di interrupt 0 è il modo dell'8080, l'interrupt 1 è una chiamata alla posizione 038H, e interrupt 2 è una chiamata indiretta la quale usa il contenuto del registro speciale 1, più 8 bit forniti dal dispositivo di interruzione come puntatore nella memoria alla routine di interrupt.

Questi modi saranno spiegati nel Capitolo 6.

Alla fine, speciali pin (piedini) sullo Z80 faranno scattare un meccanismo di interrupt che sarà anch'esso spiegato nel Capitolo 6. Sono i pin IRQ e NMI.

SOMMARIO

Ora sono state descritte le cinque categorie di istruzioni disponibili sullo Z80. I dettagli sulle istruzioni individuali sono fornite nel prossimo paragrafo del libro. Per iniziare un programma non è necessario capire il ruolo di ogni istruzione. All'inizio è sufficiente la conoscenza di alcune istruzioni essenziali di ogni tipo. Comunque, quando cominciate a scrivere programmi da soli, dovreste imparare tutto sulle istruzioni dello Z80, se volete scrivere dei buoni programmi.

Naturalmente, all'inizio, l'efficienza non è importante, ecco perché la maggior parte delle istruzioni possono essere ignorate.

Un aspetto importante non è ancora stato descritto. Questo è la serie delle tecniche di indirizzamento rese disponibili sullo Z80 per facilitare il ricupero dei dati dallo spazio di memoria. Le tecniche di indirizzamento saranno studiate nel prossimo capitolo.

LE ISTRUZIONI DELLO Z80: DESCRIZIONE INDIVIDUALE

ABBREVIAZIONI

FLAG	ON	OFF
CARRY	C (CARRY)	NC (Nessun carry)
SEGNO	M (MENO)	P (più)
ZERO	Z (ZERO)	NZ (non zero)
PARITÀ	PR (PARI)	PO (dispari)

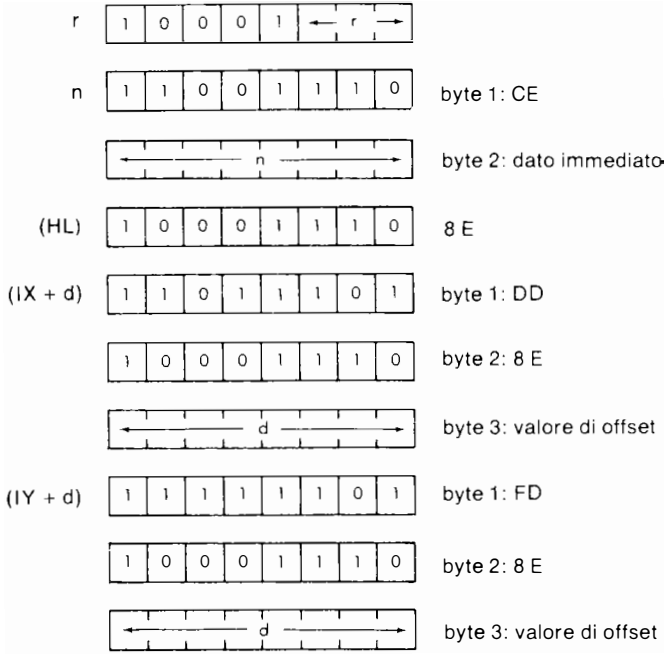
● Cambiato a secondo dell'operazione
O flag è posto a zero
1 flag è posto ad uno
? flag e posto casualmente dall'operazione
X caso speciale, vedere la nota che l'accompagna n quella pagi :

Le posizioni di bi 3 e 5 sono sempre casual .

ADC A, s Somma l'accumulatore e l'operando specificato con il riporto

Funzione: A ← A + s + C

Formato: s: può essere r, n, (HL), (IX + d), oppure (IY + d)

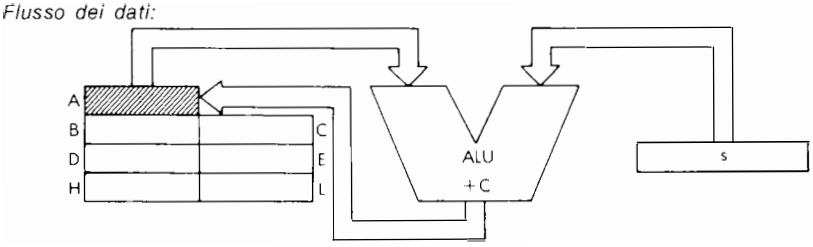


r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: L'operando s ed il flag di carry (o riporto) C dal registro di stato sono sommati all'accumulatore, ed il risultato è immagazzinato nell'accumulatore.
s è definito nella descrizione delle analoghe istruzioni ADD

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	µsec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Modo d'indirizzamento: r: implicito; n: immediato; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: ADC A, r

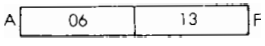
A	B	C	D	E	H	L
8F	88	89	8A	8B	8C	8D

Flag:

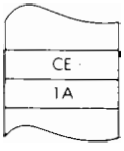
S	Z		H		P/V	N	C
●	●		●		●	○	●

Esempio: ADC A, 1 A

Prima:



Dopo



CODICE OGGETTO

ADC HL, ss

Somma con il carry HL e la coppia di registri ss.

Funzione: HL ← HL + ss + C

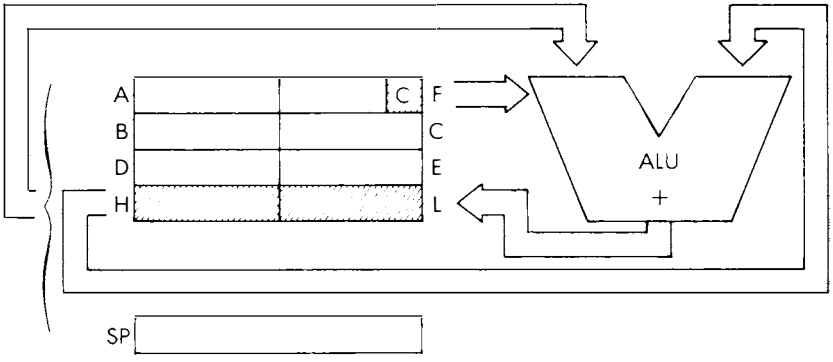
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	s	s	1	0	1	0	byte 2

Descrizione: Il contenuto della coppia di registri HL è sommato al contenuto della coppia di registri specificata, e poi è aggiunto il contenuto del flag del riporto. Il risultato finale è immagazzinato di nuovo in HL. ss può essere ognuno di questi:

BC — 00 HL — 10
DE — 01 SP — 11

Flusso dei dati:



Temporizzazione: 4 cicli M; 15 stati T: 7.5 µsec @ 2 MHz

Modo d'indirizzamento: Implicito

Codici del byte: ss:

	BC	DE	HL	SP
ED—	4A	5A	6A	7A

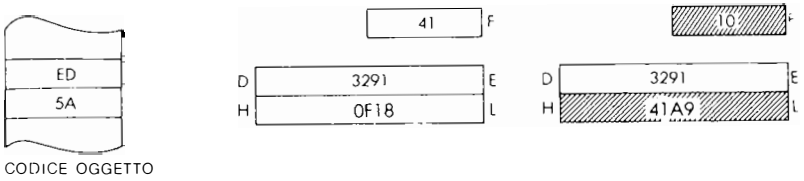
Flag:

S	Z		H		P/V	N	C
●	●		●		●	○	●

H è posto ad uno se c'è un riporto dal bit 11.

Esempio: ADC HL, DE

Prima: Dopo:



ADD A, (HL)

Somma l'accumulatore con la locazione di memoria indirizzata indirettamente (HL).

Funzione: A ← A + (HL)

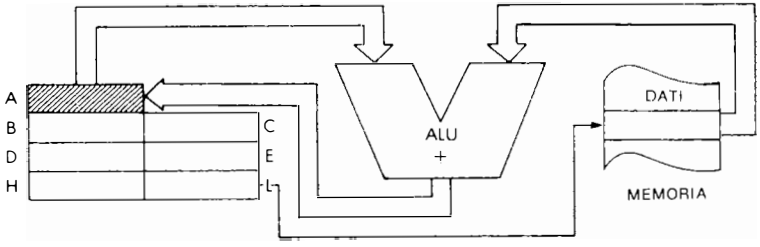
Formato:

1	0	0	0	0	1	1	0

86

Descrizione: Il contenuto dell'accumulatore è sommato al contenuto della locazione di memoria indirizzata dalla coppia di registri HL. Il risultato è immagazzinato nell'accumulatore.

Flusso dei dati:



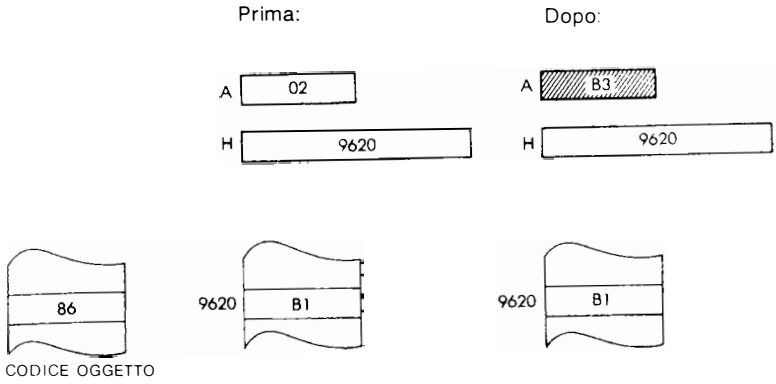
Temporizzazione: 2 cicli M; 7 stati T: 3.5 µsec @ 2 MHz

Modo d'indirizzamento: Indiretto

Flag:

S	Z		H		P/V	N	C
●	●		●		●	○	●

Esempio: ADD A, (HL)



ADD A, (IX + d) Somma l'accumulatore con la locazione di memoria indirizzata in modo indicizzato (IX + d)

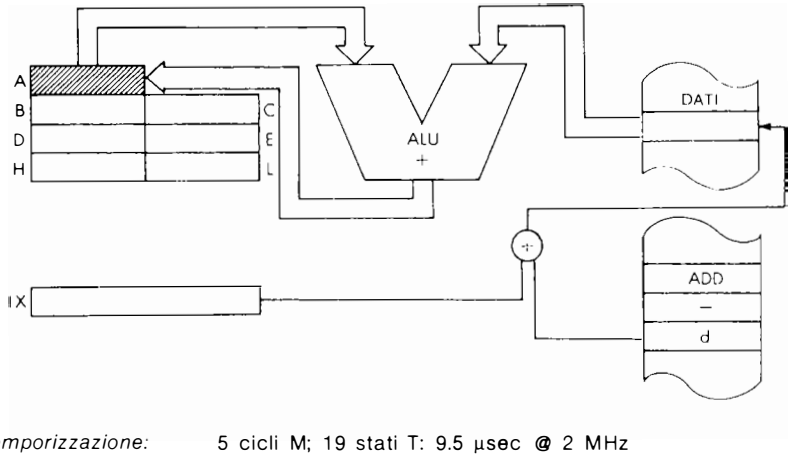
Funzione: A ← A + (IX + d)

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	0	0	0	0	1	1	0	byte 2: 86
← d →								byte 3: valore di offset

Descrizione: Il contenuto dell'accumulatore è sommato al contenuto della locazione di memoria indicizzata dal contenuto del registro IX più il valore di offset immediato. Il risultato è immagazzinato nell'accumulatore.

Flusso dei dati:



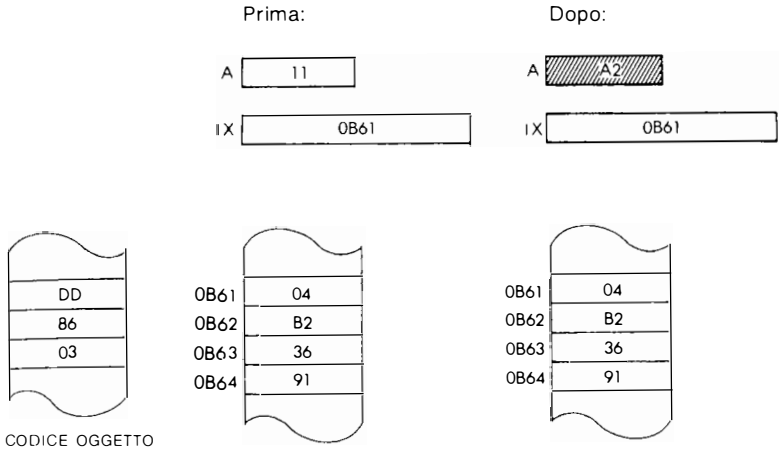
Temporizzazione: 5 cicli M; 19 stati T: 9.5 µsec @ 2 MHz

Modo d'indirizzamento: Indicizzato

Flag:

S	Z		H		P/V	N	C
●	●		●		●	○	●

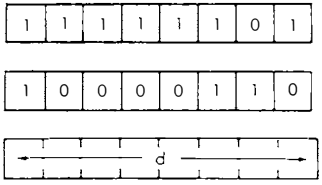
Esempio: ADD A, (IX + 3)



ADD A, (IY + d) Somma l'accumulatore con la locazione di memoria indirizzata in modo indicizzato (IY + d)

Funzione: A ← A + (IY + d)

Formato:



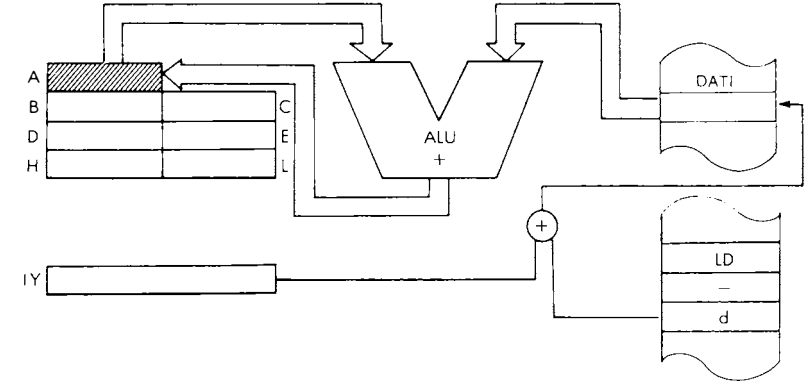
byte 1: FD

byte 2: 86

byte 3: valore di offset

Descrizione: Il contenuto dell'accumulatore è sommato al contenuto della locazione di memoria indirizzata dal contenuto del registro IY più il valore di offset assegnato. Il risultato è immagazzinato nell'accumulatore.

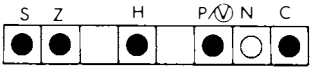
Flusso dei dati:



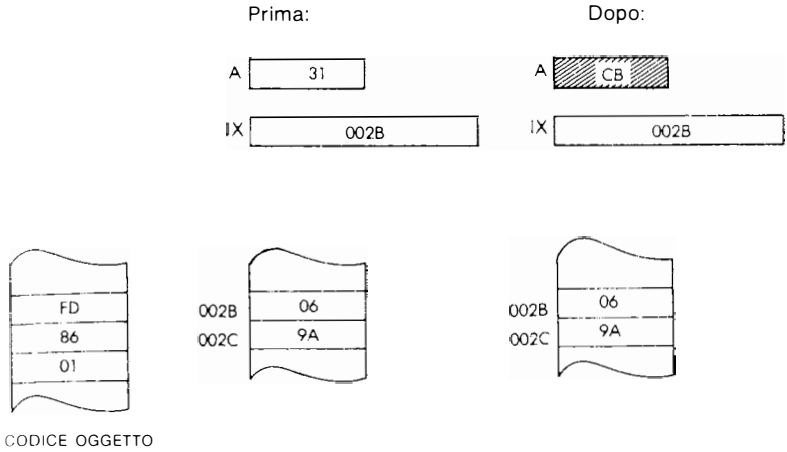
Temporizzazione: 5 cicli M; 19 stati T: 9.5 µsec @ 2 MHz

Modo d'indirizzamento: Indicizzato

Flag:



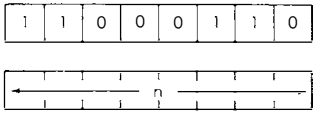
Esempio: ADD A, (IY + 1)



ADD A, n Somma l'accumulatore con il dato immediato n.

Funzione: A ← A + n

Formato:

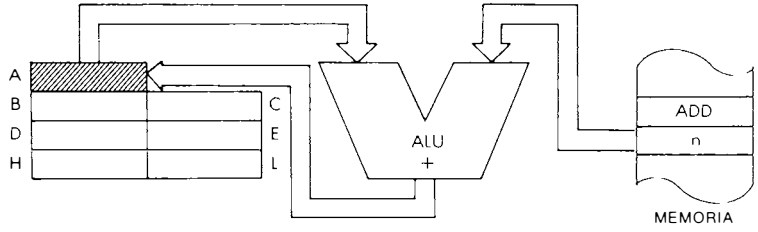


byte 1: C6

byte 2: dato immediato

Descrizione: Il contenuto dell'accumulatore è sommato al contenuto della locazione di memoria che segue immediatamente il Codice operativo. Il risultato è immagazzinato nell'accumulatore.

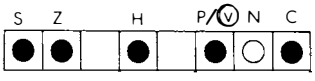
Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T: 3.5 μsec @ 2 MHz

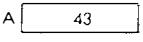
Modo d'indirizzamento: Immediato

Flag:

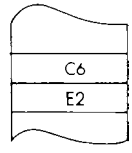


Esempio: ADD A, E 2

Prima:



Dopo:



CODICE OGGETTO

ADD A, r

Somma l'accumulatore con il registro r.

Funzione: A ← A + r

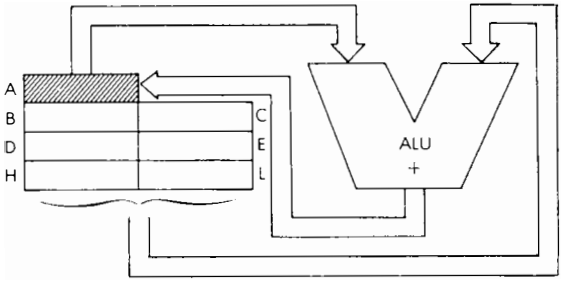
Formato:

1	0	0	0	0	← r →

Descrizione: Il contenuto dell'accumulatore è sommato con il contenuto del registro specificato. Il risultato è posto nell'accumulatore. r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T: 2 μsec @ 2 MHz

Modo d'indirizzamento: Implicito

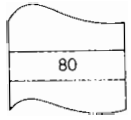
Codici del byte: r:

A	B	C	D	E	H	L
87	80	81	82	83	84	85

Flag:

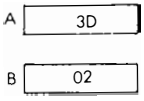
S	Z		H		P/V	N	C
●	●		●		●	○	●

Esempio: ADD A, B

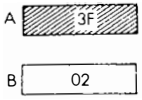


CODICE OGGETTO

Prima:



Dopo:



ADD HL, ss

Somma HL e la coppia di registri ss

Funzione: HL ← HL + ss

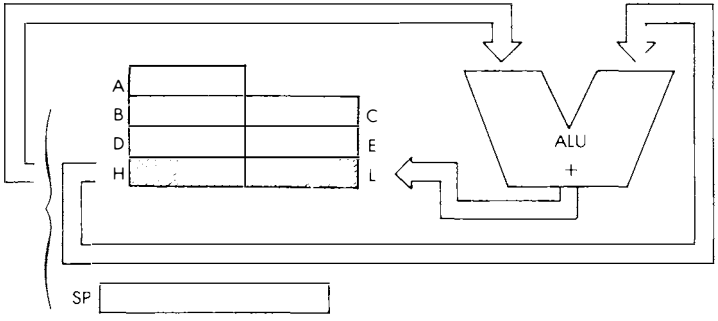
Formato:

0	0	S	S	1	0	0	1

Descrizione: Il contenuto della coppia di registri specificata è sommato al contenuto della coppia di registri HL ed il risultato è immagazzinato in HL. ss può essere ognuno di questi:

BC – 00 HL – 10
DE – 01 SP – 11

Flusso dei dati:



Temporizzazione: 3 cicli M; 11 stati T: 5.5 µsec @ 2 MHz

Modo d'indirizzamento: Implicito

Codici del byte: ss:

BC	DE	HL	SP
09	19	29	39

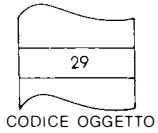
Flag:

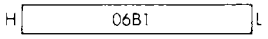
S	Z		H		P/V	N	C
			●			○	●

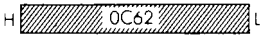
C è posto ad 1 dal riporto del bit 15, altrimenti è posto a 0. H è posto ad 1 da un riporto dal bit 11.

Esempio: ADD HL, HL

Prima: Dopo:







ADD IX, rr

Somma IX con la coppia di registri rr

Funzione: IX ← IX + rr

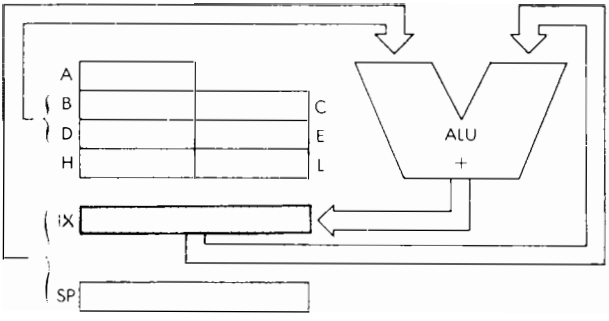
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	r	r	1	0	0	1	byte 2

Descrizione: Il contenuto del registro IX è sommato al contenuto della coppia dei registri specificato ed il risultato è immagazzinato di nuovo in IX. rr può essere ognuno di questi:

BC — 00 IX — 10
DE — 01 SP — 11

Flusso dei dati:



Temporizzazione: 4 cicli M; 15 stati T: 7.5 µsec @ 2 MHz

Modo d'indirizzamento: Implicito

Codici del byte:

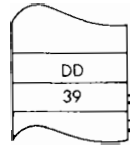
rr:	BC	DE	IX	SP
DD —	09	19	29	39

Flag:

S	Z		H		P/V	N	C
			●			○	●

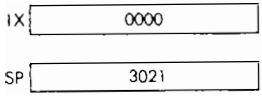
H è posto ad 1 dal riporto del bit 11. C è posto ad 1 dal riporto dal bit 15.

Esempio: ADD IX, SP

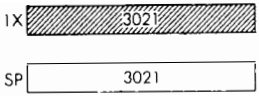


CODICE OGGETTO

Prima:



Dopo:



ADD IY, rr

Somma IY e la coppia di registri rr.

Funzione: IY ← IY + rr

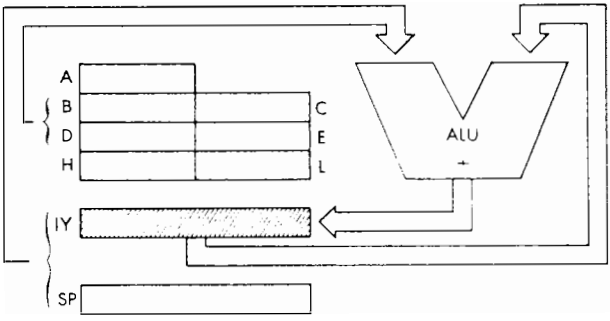
Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	r	r	1	0	0	1	byte 2

Descrizione: Il contenuto del registro IY è sommato al contenuto della coppia di registri specificato ed il risultato è immagazzinato di nuovo in IY. rr può essere ognuno di questi:

BC – 00 IY – 10
DE – 01 SP – 11

Flusso dei dati:



Temporizzazione: 4 cicli M; 15 stati T: 7.5 µsec @ 2 MHz

Modo d'indirizzamento: Implicito

Codici del byte:

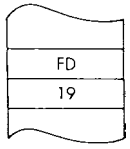
rr:	BC	DE	IY	SP
FD--	09	19	29	39

Flag:

S	Z		H		P/V	N	C
			●			○	●

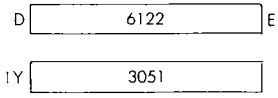
H è posto ad uno dal riporto del bit 11. C è posto ad uno dal riporto del bit 15.

Esempio: ADD IY, DE

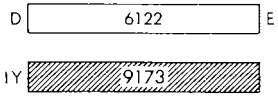


CODICE OGGETTO

Prima:



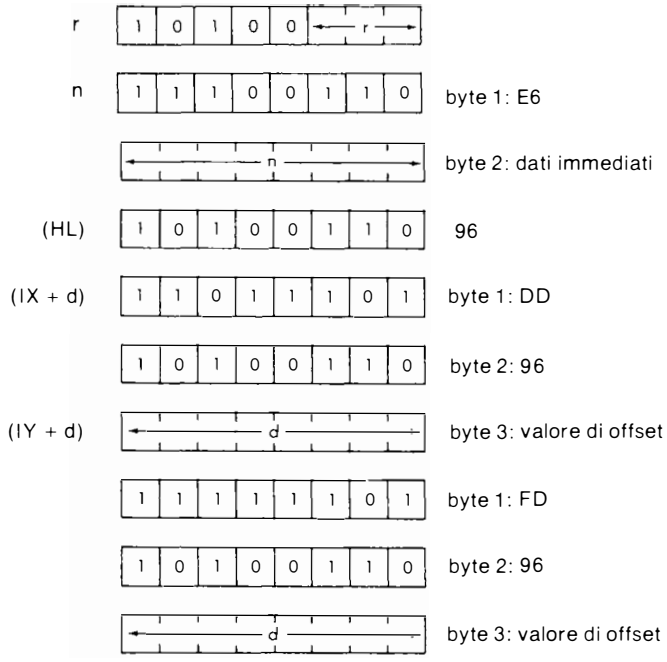
Dopo:



AND s Esegui l'AND logico fra l'accumulatore e l'operando s.

Funzione: $A \leftarrow A \wedge s$

Formato: S: può essere r, n, (HL), (IX + d) oppure (IY + d)

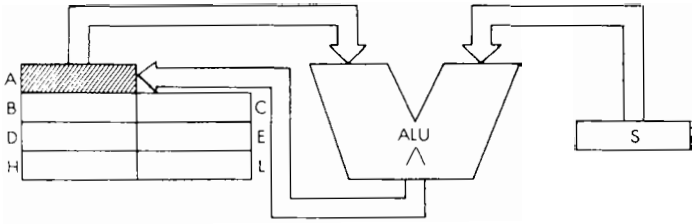


r può essere ognuno di questi:

A – 111	E – 011
B – 000	H 100
C – 011	L – 101
D – 010	

Descrizione: Viene eseguito l'AND logico tra l'accumulatore e l'operando specificato ed il risultato è immagazzinato nell'accumulatore. S è definito nella descrizione delle analoghe istruzioni ADD

Flusso dei dati:



Temporizzazione:

s:	cicli M:	stati T:	µsec @2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Modo d'indirizzamento: r: implicito; n: immediato; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: AND r

r: A	B	C	D	E	H	L
A7	A0	A1	A2	A3	A4	A5

Flag:

S	Z		H		P/V	N	C
●	●		1		●	○	○

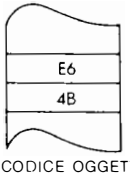
Esempio: AND 4B

Prima:

A 36

Dopo:

A 02



CODICE OGGETTO

BIT b, (HL)

Provare il bit b della locazione di memoria indirizzata indirettamente (HL)

Funzione: $Z \leftarrow \overline{(HL)_b}$

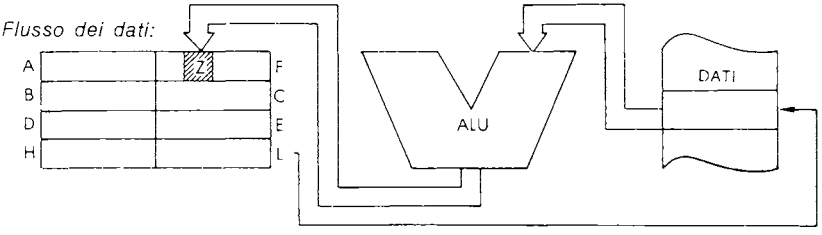
Formato:

1	1	0	0	1	0	1	1	byte 1: CB
0	1	← b →			1	1	0	byte 2

Descrizione: Il bit specificato della locazione di memoria indirizzata dal contenuto della coppia di registri HL è provato ed il flag Z è posizionato a seconda del risultato. b può essere ognuno di questi:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Flusso dei dati:



Temporizzazione: 3 cicli M; 12 stati T; 6 µsec @ 2 MHz

Modo d'indirizzamento: Indiretto

Flag:

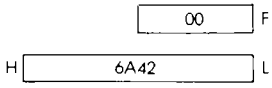
S	Z	H	P/V	N	C
?	●	1	?	0	

Codici del byte:

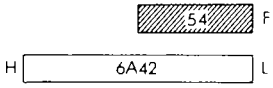
b:	0	1	2	3	4	5	6	7
CB-	46	4E	56	5E	66	6E	76	7E

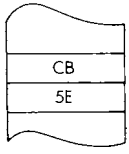
Esempio: BIT 3, (HL)

Prima:

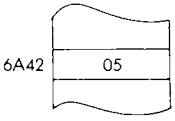


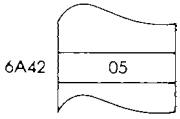
Dopo:





CODICE OGGETTO





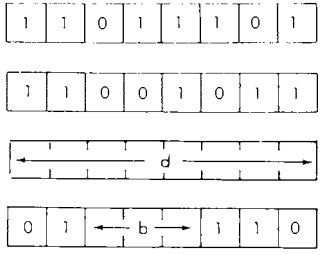
BIT b, (IX + d)

Prova il bit b della locazione di memoria indirizzata in modo indicizzato (IX + d)

Funzione:

$Z \leftarrow \overline{(IX + d)_b}$

Formato:



byte 1: DD

byte 2: CB

byte 3: valore di offset

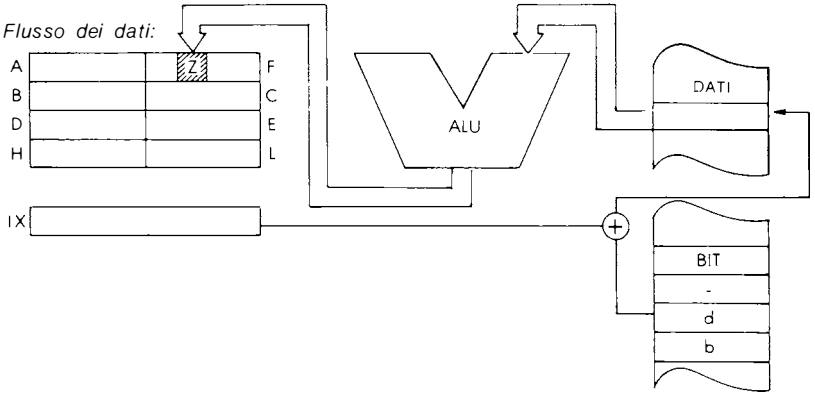
byte 4

Descrizione:

Il bit specificato della locazione di memoria indirizzata dal contenuto del registro IX più il dato valore di offset è provato ed il flag Z è posizionato a seconda del risultato. b può essere ognuno di questi:

0 – 000	5 – 101
1 – 001	6 – 110
2 – 010	7 – 111
3 – 011	
4 – 100	

Flusso dei dati:



Temporizzazione: 5 cicli M; 20 stati T: 10 µsec @ 2 MHz

Modo d'indirizzamento: Indicizzato

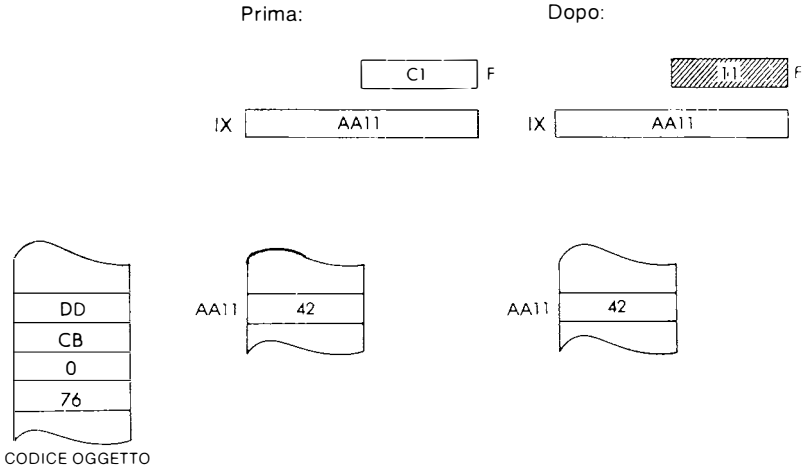
Codici del byte:

b:	0	1	2	3	4	5	6	7
DD-CB-d-	46	4E	56	5E	66	6E	76	7E

Flag:

S	Z		H		P/V	N	C
?	●		1		?	0	

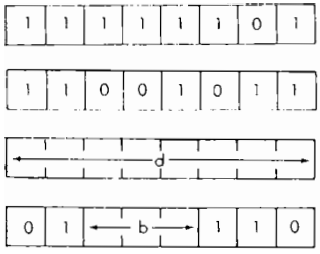
Esempio: BIT 6, (IX + 0)



BIT b, (IY + d) Prova il bit b della locazione di memoria indirizzata in modo indicizzato (IY + d).

Funzione: $Z \leftarrow \overline{(IY + d)_b}$

Formato:



byte 1: FD

byte 2: CB

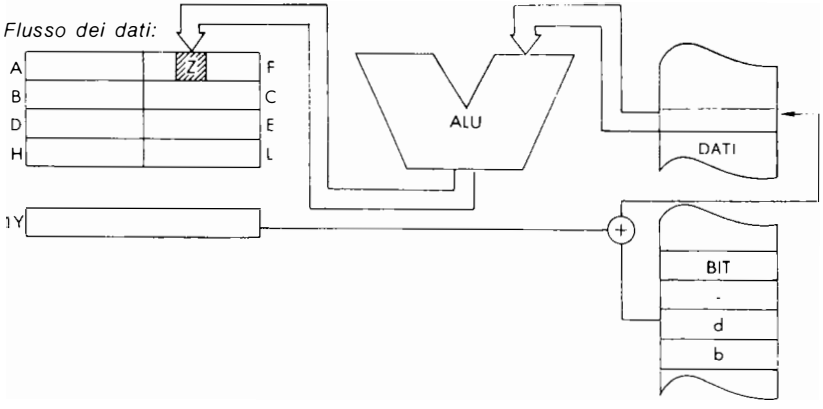
byte 3: valore di offset

byte 4

Descrizione: Il bit specificato della locazione di memoria indirizzata dal contenuto del registro IY più il dato valore di offset è provato e il flag Z è posizionato a seconda del risultato. b può essere ognuno di questi:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	

Flusso dei dati:



Temporizzazione: 5 cicli M; 20 stati T: 10 µsec @ 2 MHz

Modo d'indirizzamento: Indicizzato

Codici del byte: b:

0	1	2	3	4	5	6	7
46	4E	56	5E	66	6E	76	7E

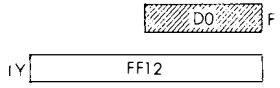
Flag:

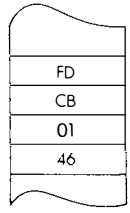
S	Z		H		P/V	N	C
?	●		1		?	0	

Esempio: BIT 0, (IY + 1)

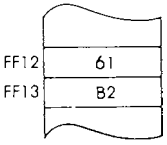
Prima:

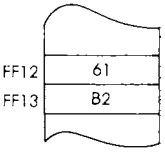
Dopo:





CODICE OGGETTO

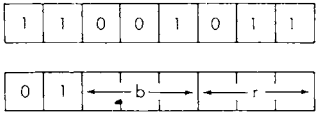




BIT b, r Prova il bit b del registro r.

Funzione: $Z \leftarrow \overline{r_b}$

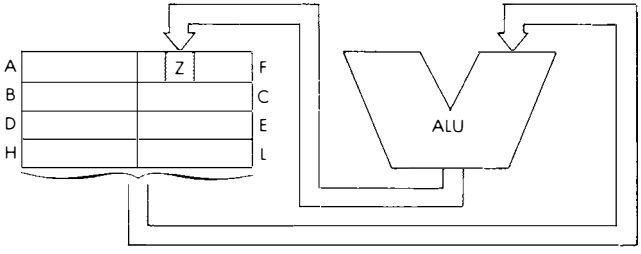
Formato:



Descrizione: Il bit specificato dell'assegnato registro è provato ed il flag zero è posizionato a seconda del risultato. b ed r possono essere ognuno di questi:

b:	0 – 000	4 – 100
	1 – 001	5 – 101
	2 – 010	6 – 110
	3 – 011	7 – 111
r:	A – 111	E – 011
	B – 000	H – 100
	C – 001	L – 101
	D – 010	

Flusso dei dati:



Temporizzazione. 2 cicli M; 8 stati T; 4 μsec @ 2 MHz

Modo d'indirizzamento: Implicito

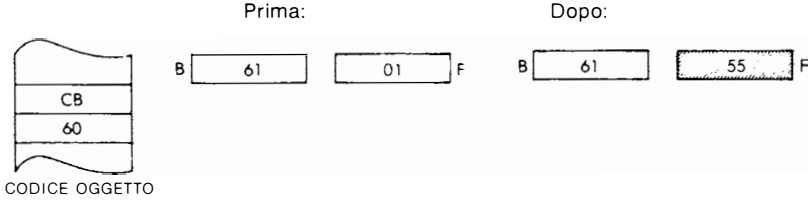
Codici del byte:

b:	r: A	B	C	D	E	H	L
0	47	40	41	42	43	44	45
1	4F	48	49	4A	4B	4C	4D
2	57	50	51	52	53	54	55
3	5F	58	59	5A	5B	5C	5D
4	67	60	61	62	63	64	65
5	6F	68	69	6A	6B	6C	6D
6	77	70	71	72	73	74	75
7	7F	78	79	7A	7B	7C	7D

Flag:

S	Z		H		P/V	N	C
?	●		1		?	0	

Esempio: BIT A, B



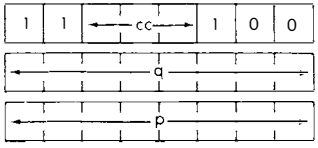
CALL cc, pq

Chiama la subroutine sotto condizione.

Funzione:

se cc è vero: $(SP - 1) \leftarrow PC_{alto}$;
$(SP - 2) \leftarrow PC_{basso}$; $SP \leftarrow SP - 2$;
$PC \leftarrow pq$
Se cc è falso: $PC \leftarrow PC + 3$

Formato:



byte 1

byte 2: indirizzo, ordine inferiore

byte 3: indirizzo, ordine superiore

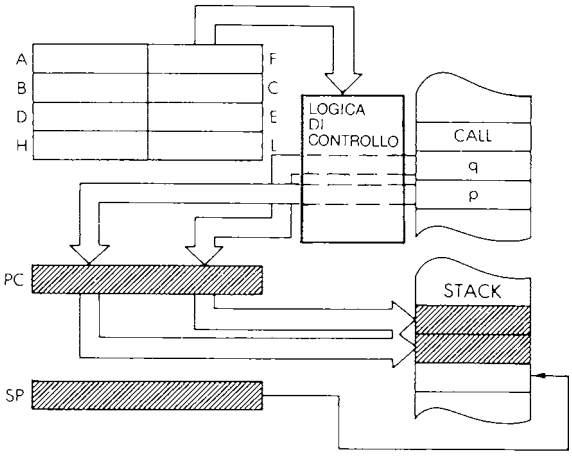
Descrizione:

Se è soddisfatta la condizione, il contenuto del contatore di programma è spinto nello stack come descritto per le istruzioni PUSH. Poi, il contenuto della locazione di memoria che segue immediatamente il codice operativo è caricato nel PC di ordine inferiore ed il contenuto della seconda locazione di memoria dopo il codice operativo è caricato nella metà di ordine superiore del PC. La prossima istruzione prelevata sarà da questo nuovo indirizzo. Se la condizione non è soddisfatta, l'indirizzo pq è ignorato ed è eseguita l'istruzione che segue. cc può essere ognuno di questi:

NZ — 000	PO — 100
Z — 001	PE — 101
NC — 010	P — 100
C — 011	M — 111

Per ripristinare il PC può essere usata una istruzione RET alla fine della subroutine che è stata chiamata.

Flusso dei dati:



Temporizzazione:

	cicli M:	*stati T:*	*µsec @ 2 MHz*
condizione vera:	*5*	*17*	*8.5*
condizione non vera:	*3*	*10*	*5*

Modo d'indirizzamento: Immediato

Codici del byte:

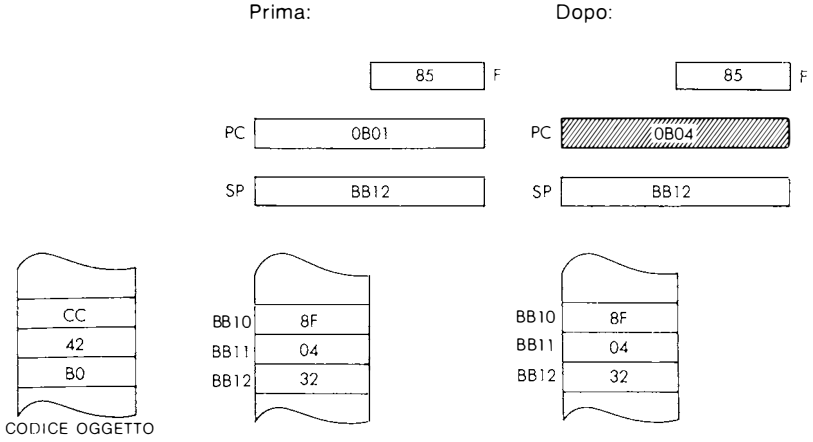
CC: NZ	Z	NC	C	PO	PE	P	M	
C4	CC	D4	DC	E4	EC	F4	FC	-q-p

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: CALL Z, B042



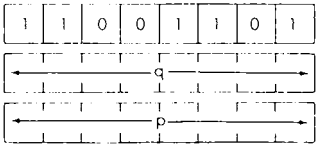
CALL p q

Chiama la subroutine che si trova alla locazione pq.

Funzione:

(SP – 1) ← PCalto ; (SP – 2) ← *PCbasso* ;
SP ← SP – 2; PC ← pq

Formato:



byte 1: CD

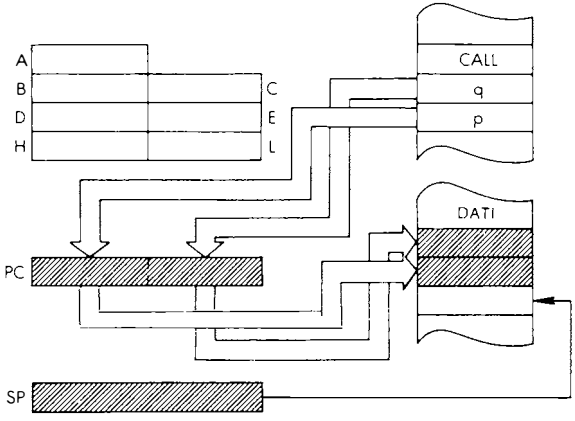
byte 2: indirizzo, ordine inferiore

byte 3: indirizzo, ordine superiore

Descrizione:

Il contenuto del contatore di programma è spinto nello stack come descritto per le istruzioni PUSH. Il contenuto della locazione di memoria che segue immediatamente il codice operativo è poi caricato nella metà di ordine inferiore del PC ed il contenuto della seconda locazione di memoria dopo il codice operativo è caricato nella metà di ordine superiore del PC. L'istruzione successiva sarà prelevata da questo nuovo indirizzo.

Flusso dei dati:



Temporizzazione: 5 cicli M; 17 stati T; 8.5 µsec @ 2 MHz

Modo d'indirizzamento: Immediato

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

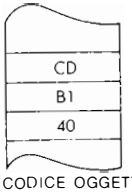
Esempio: CALL 40B1

Prima:

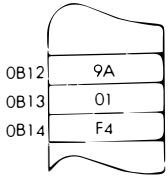
PC	AA40
SP	0B14

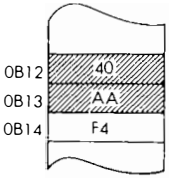
Dopo:

PC	40B1
SP	0B12



CODICE OGGETTO





CCF Complementa il flag di riporto (carry).

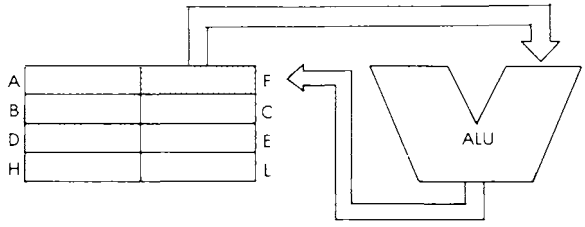
Funzione: $C \leftarrow \overline{C}$

Formato:

0	0	1	1	1	1	1	1	3 F

Descrizione: Il flag di riporto è completato.

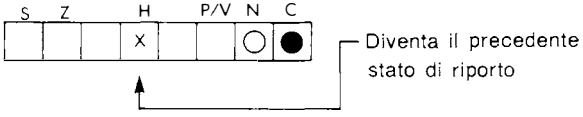
Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T: 2 μsec @ 2 MHz

Modo d'indirizzamento: Implicito

Flag:



CP s Confronta l'operando s con l'accumulatore.

Funzione: A — s

Formato: s: può essere n, (HL), (IX + d), oppure (IY + d).

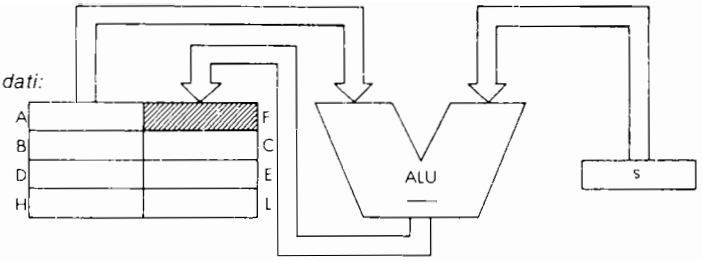
r	1	0	1	1	1	← r →				
n	1	1	1	1	1	1	1	0	FE	
	← n →								byte 2: dati immediati	
(HL)	1	0	1	1	1	1	1	0	byte 1: BE	
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD	
	1	0	1	1	1	1	1	0	byte 2: BE	
	← d →								byte 3: valore di offset	
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD	
	1	0	1	1	1	1	1	0	byte 2: BE	
	← d →								byte 3: valore di offset	

r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: L'operando specificato è sottratto dall'accumulatore, ed il risultato è scartato. s è definito nella descrizione delle analoghe istruzioni ADD.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Modo d'indirizzamento: r: implicito; n: immediato; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: CP r:

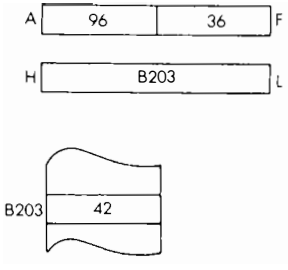
r:	A	B	C	D	E	H	L
	BF	B8	B9	BA	BB	BC	BD

Flag:

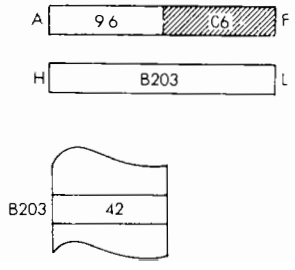
S	Z		H		P/V	N	C
●	●		●		●	1	●

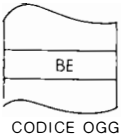
Esempio: CP (HL)

Prima:



Dopo:





CODICE OGGETTO

CPD

Confronta e decrementa.

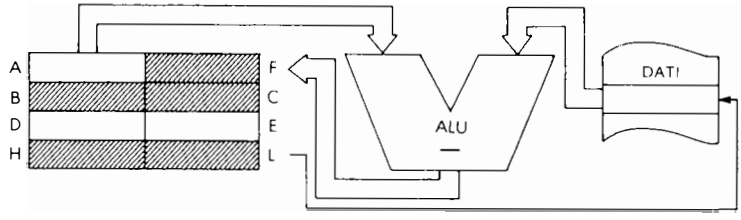
Funzione: A − [HL]; HL ← HL − 1; BC ← BC − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	0	1	byte 2: A9

Descrizione: Il contenuto della locazione di memoria indirizzato dalla coppia di registri HL è sottratto dal contenuto dell'accumulatore ed il risultato è scartato. Poi sono decrementati sia la coppia di registri HL che la coppia di registri BC.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T: 8 µsec @ 2 MHz

Modo d'indirizzamento: indiretto.

Flag:

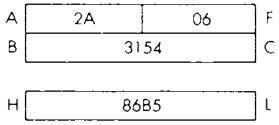
S	Z	H	P/V	N	C
●	X	●	X	1	

Posto a 0 se è BC = 0 dopo l'esecuzione; altrimenti posto ad 1

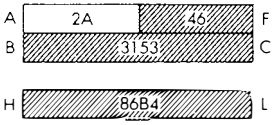
Posto ad 1 se è A = [HL]

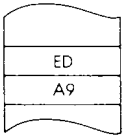
Esempio: CPD

Prima:

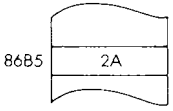


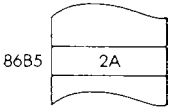
Dopo:





CODICE OGGETTO





CPDR

Confronta il blocco e decrementa.

Funzione:

A − [HL]; HL ← HL − 1; BC ← BC − 1;
Ripeti fino a BC = 0 oppure A = [HL]

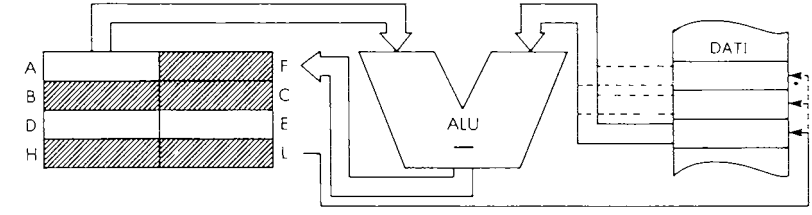
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	0	1	byte 2: B9

Descrizione:

Il contenuto della locazione di memoria indirizzata dalla coppia di registri HL è sottratto dal contenuto dell'accumulatore ed il risultato è scartato. Poi sono decrementati sia la coppia di registri BC che la coppia di registri HL. Se BC = 0 ed A = HL, il contatore di programma è decrementato di due e l'istruzione è rieseguita.

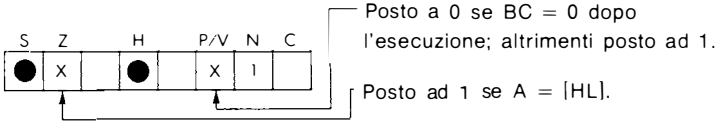
Flusso dei dati:



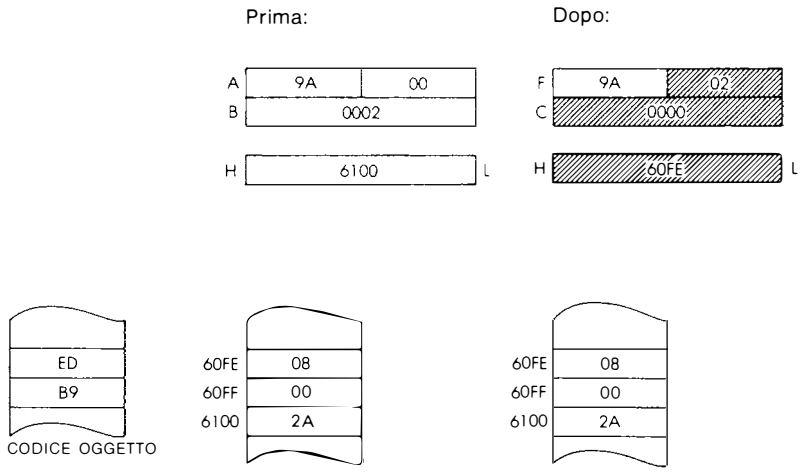
Temporizzazione:

BC = 0 oppure A = [HL]: 4 cicli M; 16 stati T: 8 µsec @ 2 MHz
BC = 0 ed A = [HL]: 5 cicli M; 21 stati T: 10.5 µsec @ 2 MHz

Flag:



Esempio: CPDR



CPI

Confronta ed incrementa.

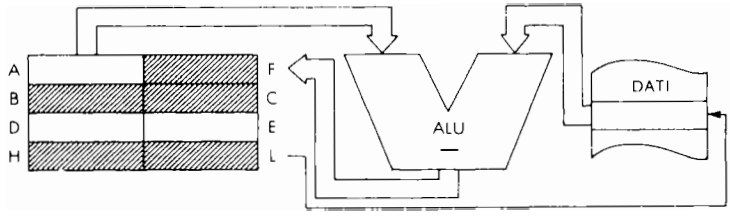
Funzione: A − [HL]; HL ← HL + 1; BC ← BC − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	0	1	byte 2: A1

Descrizione: Il contenuto della locazione di memoria indirizzata dalla coppia di registri HL è sottratto dal contenuto dell'accumulatore ed il risultato è scartato. La coppia di registri HL è incrementata e la coppia di registri BC è decrementata.

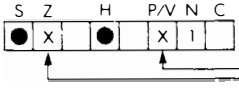
Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T: 8 µsec @ 2 MHz.

Modo d'indirizzamento: indiretto.

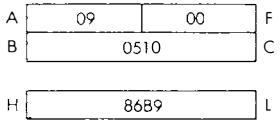
Flag:



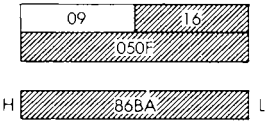
Posto a 0 se BC = 0 dopo l'esecuzione altrimenti posto ad 1
Posto ad 1 se A = [HL]

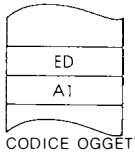
Esempio: CPI

Prima:

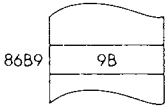


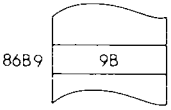
Dopo:





CODICE OGGETTO





CPIR

Confronta il blocco ed incrementa.

Funzione:

A − |HL|; H ← HL + 1; BC ← BC − 1;
Ripetere fino a che BC = 0 oppure A = |HL|

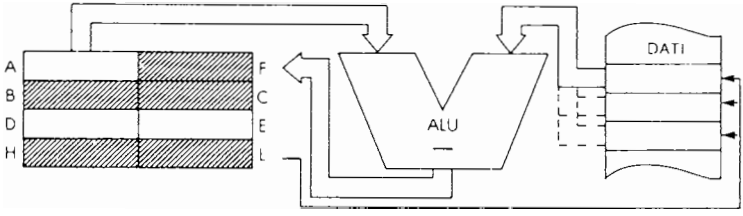
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	0	1	byte 2: B1

Descrizione:

Il contenuto della locazione di memoria indirizzata dalla coppia di registri HL è sottratto dal contenuto dell'accumulatore ed il risultato è scartato. Poi, la coppia di registri HL è incrementata ed è decrementata la coppia di registri BC. Se BC ≠ 0 e A ≠ |HL|, allora il contatore di programma è decrementato di due ed è rieseguita l'istruzione.

Flusso dei dati:

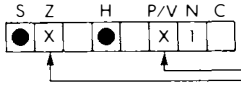


Temporizzazione:

BC = 0 oppure A = |HL|: 4 cicli M; 16 stati T: 8 μsec @ 2 MHz
BC ≠ 0 e A ≠ |HL|: 5 cicli M; 21 stati T: 10,5 μsec @ 2 MHz

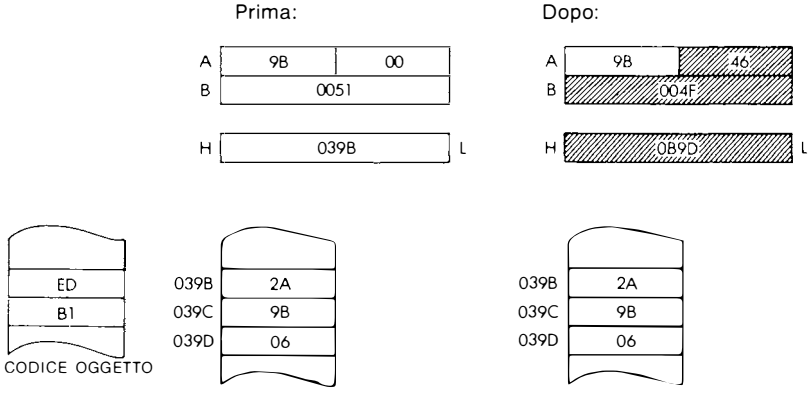
Modo d'indirizzamento: indiretto.

Flag:



Posto a 0 se BC = 0 dopo l'esecuzione; altrimenti posto ad 1
Posto a 1 se A = [HL]

Esempio: CPIR



CPL

Complementa l'accumulatore.

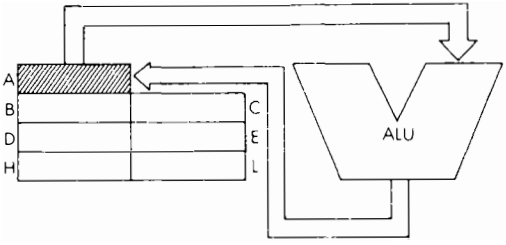
Funzione: $A \leftarrow \overline{A}$

Formato:

0	0	1	0	1	1	1	1	2F

Descrizione: Il contenuto dell'accumulatore è complementato, o invertito, e il risultato è immagazzinato di nuovo nell'accumulatore (complemento ad uno).

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

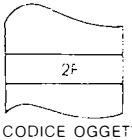
Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C
			1			1	

Esempio: CPL

Prima:



CODICE OGGETTO

Dopo:

DAA Aggiustamento decimale dell'accumulatore.

Funzione: Vedere di seguito.

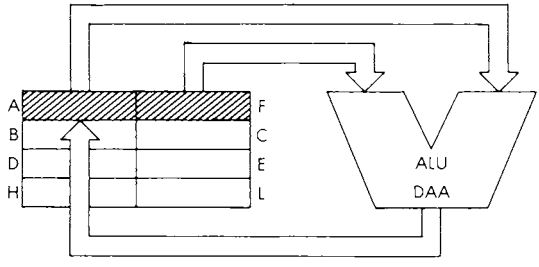
Formato:

0	0	1	0	0	1	1	1	27

Descrizione: L'istruzione somma condizionalmente "6" al nibble destro e/o sinistro dell'accumulatore, basandosi sul registro di stato, per la conversione BCD dopo le operazioni aritmetiche.

N	*C*	*valore del nibble alto*	*H*	*valore del nibble basso*	*numero sommato ad A*	*C dopo l'esecuzione*
0	0	0-9	0	0-9	00	0
(ADD,	0	0-8	0	A-F	06	0
ADC,	0	0-9	1	0-3	06	0
INC)	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
1	0	0-9	0	0-9	00	0
(SUB,	0	0-8	1	6-F	FA	0
SBC,	1	7-F	0	0-9	AO	1
DEC,	1	6-F	1	6-F	9A	1
NEG)						

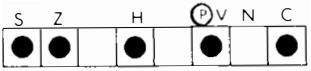
Flusso dei dati:



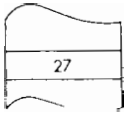
Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

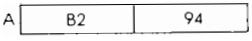


Esempio: DAA



CODICE OGGETTO

Prima:



Dopo:

DEC m Decrementa l'operando m.

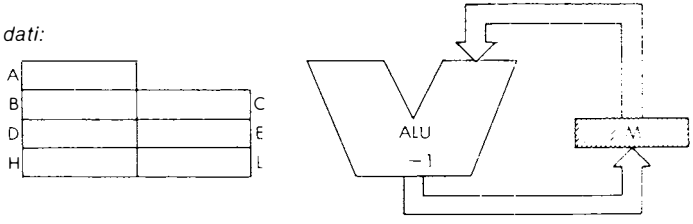
Funzione: m ← m − 1

Formato: m: può essere r, (HL), (IX + d), (IY + d)

r	0	0	← r →			1	0	1	
(HL)	0	0	1	1	0	1	0	1	35
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	0	0	1	1	0	1	0	1	byte 2: 35
	← d →								byte 3: valore di offset
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	0	0	1	1	0	1	0	1	byte 2: 35
	← d →								byte 3: valore di offset

Descrizione: Il contenuto della locazione indirizzata dall'operando specifico è decrementato e immagazzinato di nuovo in quella locazione. m è definito nella descrizione delle analoghe istruzioni INC.

Flusso dei dati:



Temporizzazione:

m:	*cicli M:*	*stati T:*	µsec @ *MHz:*
r	1	4	2
(HL)	3	11	5.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r implicito; (HL): indiretto; (IX + d), (IY + d): indicizzati.

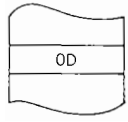
Codici del byte: DEC r r:

A	B	C	D	E	H	L
3D	05	0D	15	1D	25	2D

Flag:

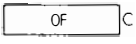
S	Z		H		P/V	N	C
●	●		●		●	1	

Esempio: DEC C



CODICE OGGETTO

Prima:



Dopo

DEC rr

Decrementa la coppia di registro rr.

Funzione: rr ← rr − 1

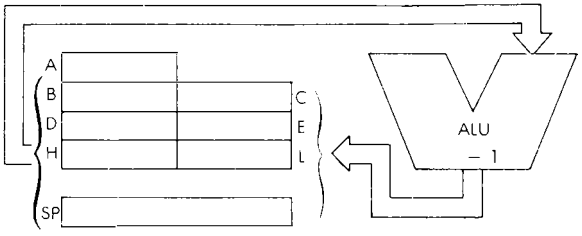
Formato:

0	0	r	r	1	0	1	1

Descrizione: Il contenuto della coppia di registri specificata è decrementato e il risultato è immagazzinato di nuovo nella coppia di registri. rr può essere ognuno di questi:

BC – 00 HL – 10
DE – 01 SP – 11

Flusso dei dati:



Temporizzazione: 1 ciclo M; 6 stati T; 3 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Codici del byte: rr:

BC	DE	HL	SP
0B	1B	2B	3B

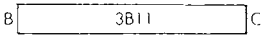
Flag:

S	Z		H		P/V	N	C

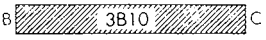
(nessun effetto)

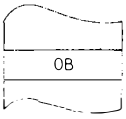
Esempio: DEC BC

Prima:



Dopo:





CODICE OGGETTO

DEC IX

Decrementa IX.

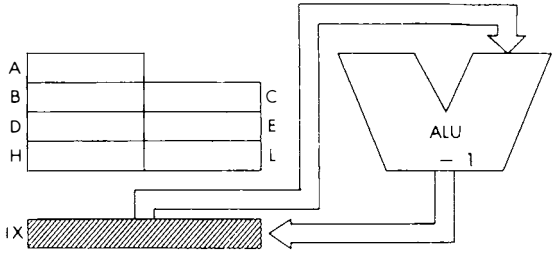
Funzione: IX ← IX − 1

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	1	0	1	1	byte 2: 2B

Descrizione: Il contenuto del registro IX è decrementato ed il risultato è immagazzinato di nuovo in IX.

Flusso dei dati:



Temporizzazione: 2 cicli M; 10 stati T; 5 μsec @ 2 MHz.

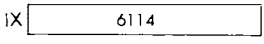
Modo d'indirizzamento: Implicito.

Flag:

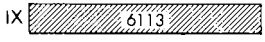
S	Z		H		P/V	N	C	
								(nessun effetto)

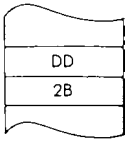
Esempio: DEC IX

Prima:



Dopo:





CODICE OGGETTO

DEC IY

Decrementa IY.

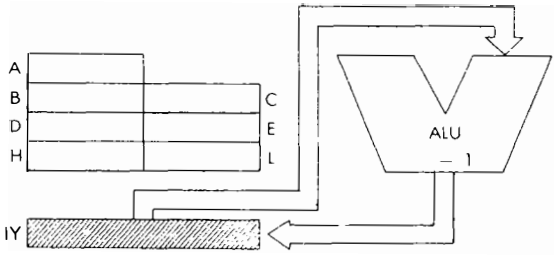
Funzione: IY ← IY − 1

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	1	0	1	1	byte 2: 2B

Descrizione: Il contenuto del registro IY è decrementato ed il risultato è immagazzinato di nuovo in IY.

Flusso dei dati:



Temporizzazione: 2 cicli M; 10 stati T; 5 µsec @ 2 MHz.

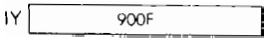
Modo d'indirizzamento: Implicito.

Flag:

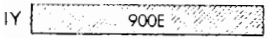
S	Z		H		P/V	N	C	
								(nessun effetto)

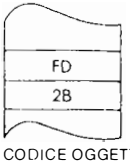
Esempio: DEC IY

Prima:



Dopo:





CODICE OGGETTO

DI Disabilita gli interrupt.

Funzione: IFF ← 0

Formato:

1	1	1	1	0	0	1	1	F3

Descrizione: I flip-flop d'interrupt sono posti a zero, perciò disabilitano gli interrupt mascherabili. Un interrupt mascherabile può essere disabilitato durante la sua esecuzione da DI. È riabilitato da un'istruzione EI.

Temporizzazione: 1 ciclo M; 4 stati T; 2 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

DJNZ e Decrementa B e salta ad e in modo relativo se B non è zero.

Funzione: B ← b − 1; se B ≠ 0: PC ← PC + e

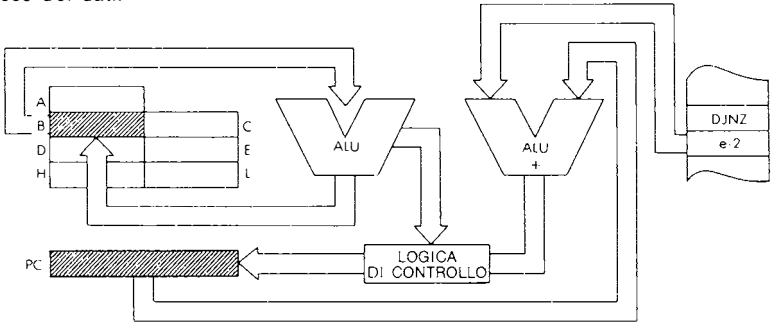
Formato:

byte 1: 10

byte 2: valore di offset

Descrizione: Il registro B è decrementato. Se il risultato non è zero, il valore di offset immediato è sommato al contatore di programma usando l'aritmetica in complemento a due così da permettere sia salti in avanti che indietro. Il valore di offset è sommato al valore di PC + 2 (dopo il salto). Come risultato, l'offset effettivo è da −126 a +129 byte. L'assemblatore sottrae automaticamente dalla sorgente il valore di offset per generare il codice esadecimale.

Flusso dei dati:



Temporizzazione: B ≠ 0: 3 cicli M; 13 stati T; 6.5 μsec @ 2 MHz.
B = 0: 2 cicli M; 8 stati T; 4 μsec @ 2 MHz.

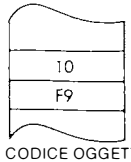
Modo d'indirizzamento: Immediato.

Flag:

S	Z		H		P/V	N	C

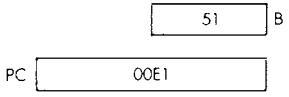
(nessun effetto)

Esempio: DJNZ $ – 5 ($ = PC corrente)

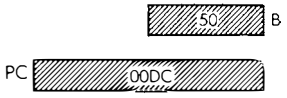


CODICE OGGETTO

Prima:



Dopo:



EI

Abilita gli interrupt.

Funzione: IFF ← 1

Formato:

1	1	1	1	1	0	1	1	FB

Descrizione: I flip-flop d'interrupt sono posti ad uno, perciò abilitando gli interrupt mascherabili dopo l'esecuzione dell'istruzione che segue l'istruzione EI. Allo stesso tempo gli interrupt mascherabili sono disabilitati.

Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: Una sequenza solita alla fine di una routine di interrupt è:
EI
RETI
L'interrupt mascherabile è riabilitato seguendo il completamento di RETI.

EX AF, AF'

Scambia accumulatore e flag con registri alternativi.

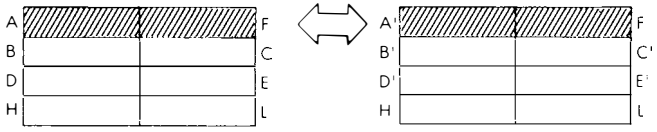
Funzione: AF ↔ AF'

Formato:

0	0	0	0	1	0	0	0	08

Descrizione: Il contenuto dell'accumulatore e del registro di stato è scambiato con il contenuto dell'accumulatore e del registro di stato alternativi.

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z	H	P/V	N	C

(nessun effetto)

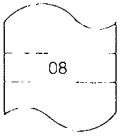
Esempio: EX AF, AF'

Prima:

A	04	81	F
A'	90	3A	F'

Dopo:

A	90	3A	F
A'	04	81	F'



CODICE OGGETTO

EX DE, HL

Scambia i registri HL e DE.

Funzione: DE ⟷ HL

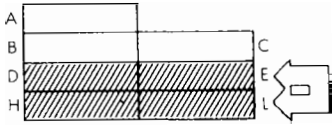
Formato:

1	1	1	0	1	0	1	1

EB

Descrizione: È scambiato il contenuto delle coppie di registro DE ed HL.

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

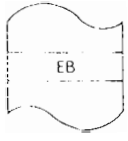
Modo d'indirizzamento: Implicito

Flag:

S	Z		H		P/V	N	C

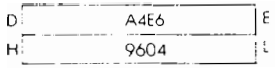
(nessun effetto)

Esempio: EX DE, HL



CODICE OGGETTO

Prima:



Dopo:

EX (SP), HL

Scambia HL con la parte alta dello stack.

Funzione: (SP) ↔ L; (SP + 1) ← H

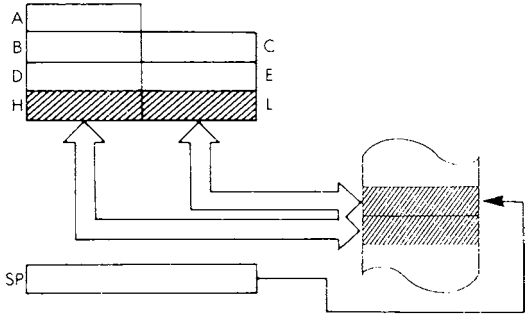
Formato:

1	1	1	0	0	0	1	1

E3

Descrizione: Il contenuto del registro L è scambiato con il contenuto della locazione di memoria indirizzata dal puntatore dello stack. Il contenuto del registro H è scambiato con il contenuto della locazione di memoria che segue immediatamente quello indirizzato dal puntatore dello stack.

Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 µsec @ 2 MHz.

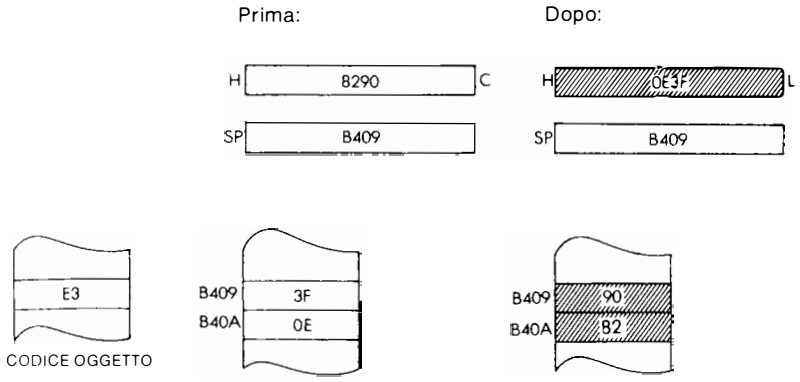
Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: EX (SP), HL



EX (SP), IX

Scambia IX con la sommità dello stack.

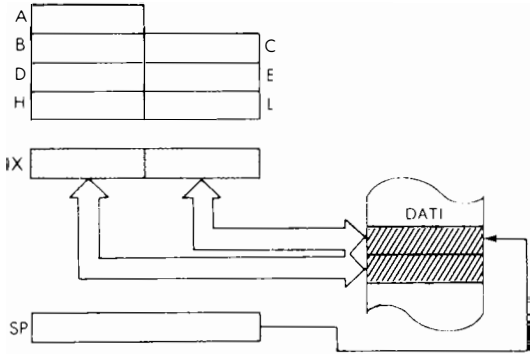
Funzione: (SP) ↔ IX$_{basso}$; (SP + 1) ↔ IX$_{alto}$

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	0	1	1	byte 2: E3

Descrizione: Il contenuto di ordine inferiore del registro IX è scambiato con il contenuto della locazione di memoria indirizzata dal puntatore di stack. Il contenuto di ordine superiore del registro IX è scambiato con il contenuto della locazione di memoria che segue immediatamente quella indirizzata dal puntatore di stack.

Flusso dei dati:



Temporizzazione: 6 cicli M; 23 stati T; 11.5 µsec @ 2 MHz.

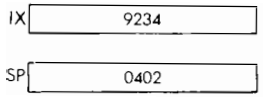
Modo d'indirizzamento: Indiretto.

Flag:

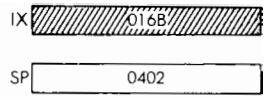
S	Z		H		P/V	N	C	
								(nessun effetto)

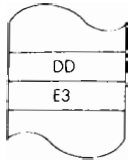
Esempio: EX (SP), IX

Prima:

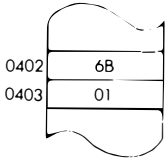


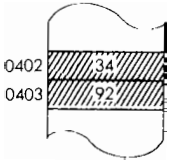
Dopo:





CODICE OGGETTO





EX (SP), IY

Scambia IY con la parte alta dello stack.

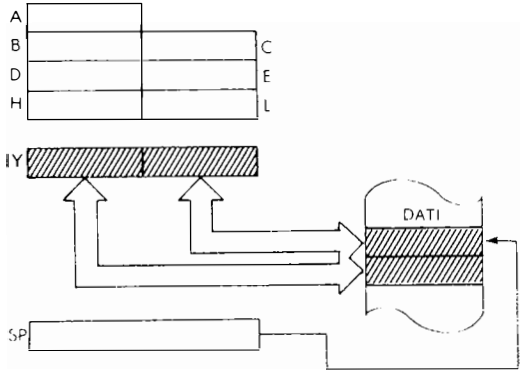
Funzione: (SP) ↔ IY_{basso}; (SP + 1) ↔ IY_{alto}.

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	0	1	1	byte 2: E3

Descrizione: Il contenuto di ordine inferiore del registro IY è scambiato con il contenuto della locazione di memoria indirizzata dal puntatore di stack. Il contenuto di ordine superiore del registro IY è scambiato con il contenuto della locazione di memoria che segue immediatamente quella indirizzata dal puntatore di stack.

Flusso dei dati:



Temporizzazione: 6 cicli M; 23 stati T; 11.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

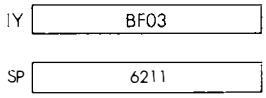
Flag:

S	Z		H		P/V	N	C

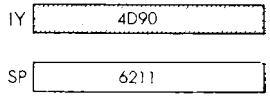
(nessun effetto)

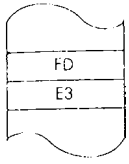
Esempio: EX (SP), IY

Prima:

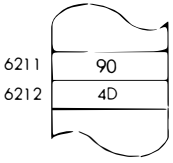


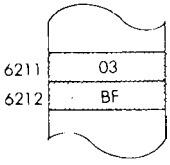
Dopo:





CODICE OGGETTO





EXX

Scambia con i registri alternativi.

Funzione: BC ↔ BC'; DE ↔ DE'; HL ↔ HL'

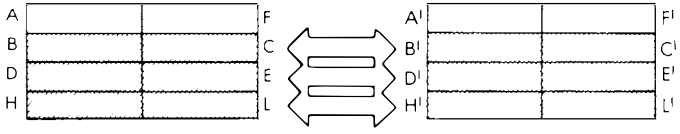
Formato:

1	1	0	1	1	0	0	1

D9

Descrizione: Il contenuto dei registri per scopi generali è scambiato con il contenuto dei corrispondenti registri alternativi.

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: EXX

Prima:

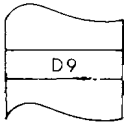
A	04	2B	F
B	39	26	C
D	54	02	E
H	F1	D0	L

A'	3F	2A	F'
B'	8C	00	C'
D'	93	D0	E'
H'	4F	E3	L'

Dopo:

A	04	2B	F
B	8C	00	C
D	93	D0	E
H	4F	E3	L

A'	3F	2A	F'
B'	39	26	C'
D'	54	02	E'
H'	F1	D0	L'



CODICE OGGETTO

HALT

Ferma la CPU.

Funzione: CPU sospesa.

Formato:

0	1	1	1	0	1	1	0

76

Descrizione: La CPU sospende il suo funzionamento ed esegue i NOP cosi da continuare i cicli di rinfresco di memoria, fino a quando è ricevuto l'interrupt o il reset.

Temporizzazione: 1 ciclo M; 4 stati T; 2 μsec @ 2 MHz + NOP indefiniti.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

IM 0

Imponi la condizione di modo d'interrupt 0.

Funzione: Controllo interno d'interrupt.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	1	0	byte 2: 46

Descrizione: Predispone il modo d'interrupt 0. In questa condizione, il dispositivo d'interrupt può inserire una istruzione sul bus dei dati per l'esecuzione, il cui primo byte deve avvenire durante il ciclo di riconoscimento dell'interrupt.

Temporizzazione: 2 cicli M; 8 stati T; 4 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

IM 1

Predisponi il modo d'interrupt 1.

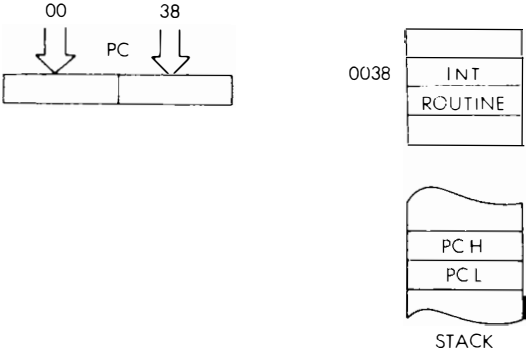
Funzione: Controllo d'interrupt interno.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	0	1	1	0	byte 2: 56

Descrizione: Predispone il modo d'interrupt 1. Quando avviene un interrupt sarà eseguita una istruzione RST 0038 H.

Flusso dei dati:



Temporizzazione: 2 cicli M; 8 stati T; 4 µsec @ 2 MHz

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

IM 2

Predisponi il modo d'interrupt 2.

Funzione: Controllo d'interrupt interno.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	1	1	1	0	byte 2: 5E

Descrizione: Predispone il modo d'interrupt 2. Quando avviene un interrupt, deve essere fornito un byte di dati dal periferico che viene usato come l'ordine inferiore di un indirizzo. L'ordine superiore di questo indirizzo di vettore è preso dal contenuto del registro I. Questo punta ad un secondo indirizzo immagazzinato in memoria, il quale è caricato nel PC ed inizia l'esecuzione.

Temporizzazione: 2 cicli M; 8 stati T, 4 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

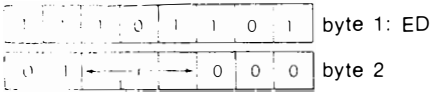
S	Z		H		P/V	N	C

IN r, (C)

Carica il registro r dalla porta (C).

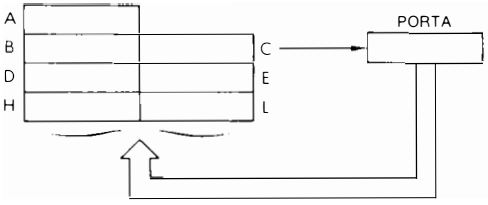
Funzione: r ← (C)

Formato:



Descrizione: Il dispositivo periferico indirizzato dal contenuto del registro C è letto ed il risultato è caricato nel registro specificato.
C fornisce i bit da A0 ad A7 del bus degli indirizzi.
B fornisce i bit da A8 ad A15.

Flusso dei dati:



r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

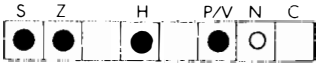
Temporizzazione: 3 cicli M; 12 stati T; 6 µsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Codici del byte:

r:	A	B	C	D	E	H	L
ED	78	40	48	50	58	60	68

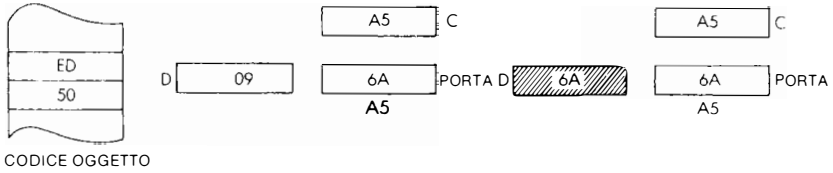
Flag:



È importante notare che INA, (N) non ha nessun effetto sui flag, mentre ne ha IN r, (C).

Esempio: IN D, (C)

Prima: Dopo:

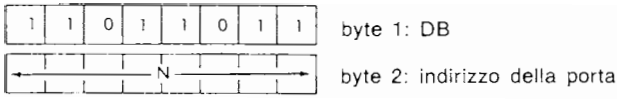


IN A, (N)

Carica l'accumulatore dalla porta d'ingresso N.

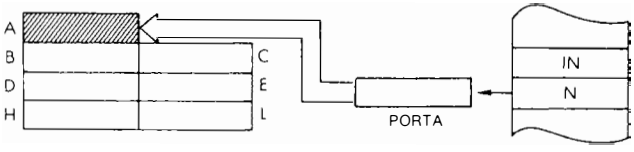
Funzione: A ← (N)

Formato:



Descrizione: Il dispositivo periferico N è letto e il risultato è caricato nell'accumulatore.
Il letterale N è posto sulle linee da A0 ad A7 del bus degli indirizzi.
A fornisce i bit da A8 ad A15.

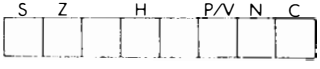
Flusso dei dati:



Temporizzazione: 3 cicli M; 11 stati T; 5.5 µsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

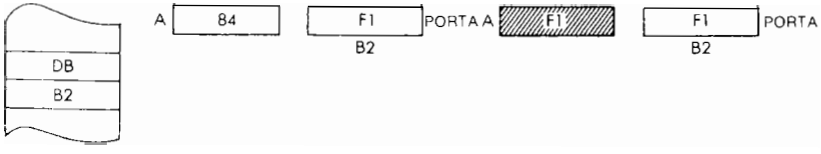
Flag:



(nessun effetto)

Esempio: IN A, (B2)

Prima: Dopo:



INC r

Incrementa il registro r.

Funzione: r ← r + 1

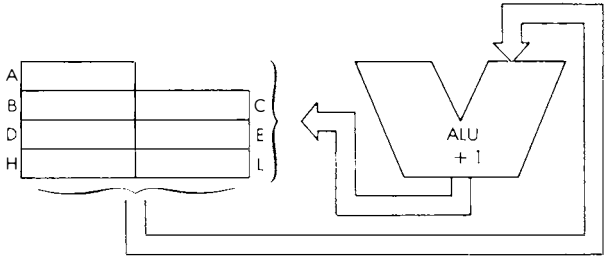
Formato:

0	0	← r →	1	0	0

Descrizione: È incrementato il contenuto del registro specificato. r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

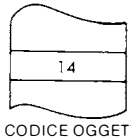
Codici del byte:

r:	A	B	C	D	E	H	L
	3C	04	0C	14	1C	24	2C

Flag:

S	Z		H		P/V	N	C
●	●		●		●	○	

Esempio: INC D



CODICE OGGETTO

Prima:

D 06

Dopo:

D 07

INC rr

Incrementa la coppia di registri rr.

Funzione: rr ← rr + 1

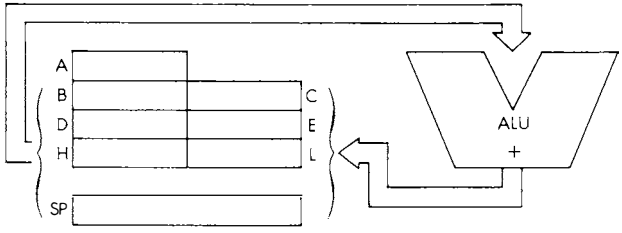
Formato:

0	0	r	r	0	0	1	1

Contenuto: Viene incrementato il contenuto della coppia di registri specificata ed il risultato viene immagazzinato di nuovo nella coppia di registri. rr può essere ognuno di questi:

BC – 00 HL – 10
DE – 01 SP – 11

Flusso dei dati:



Temporizzazione: 1 ciclo M; 6 stati T; 3 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Codici del byte: rr:

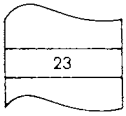
BC	DE	HL	SP
03	13	23	33

Flag:

S	Z		H		P/V	N	C

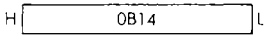
(nessun effetto)

Esempio: INC HL

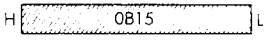


CODICE OGGETTO

Prima:



Dopo:



INC (HL)

Incrementa la locazione di memoria indirizzata indirettamente (HL).

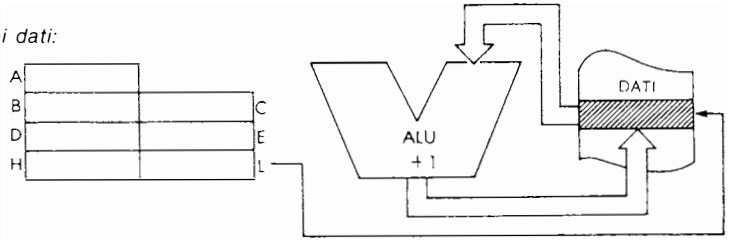
Funzione: (HL) ← (HL) + 1

Formato:

0	0	1	1	0	1	0	0	34

Descrizione: Il contenuto della locazione di memoria indirizzata dalla coppia di registri HL è incrementato e immagazzinato di nuovo in quella posizione.

Flusso dei dati:



Temporizzazione: 3 cicli M; 11 stati T; 5.5 μsec @ 2 MHz.

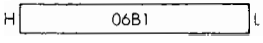
Modo d'indirizzamento: Indiretto.

Flag:

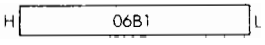
S	Z		H		P/V	N	C
●	●		●		●	○	

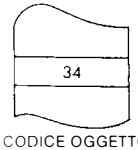
Esempio: INC (HL)

Prima:

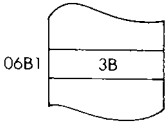


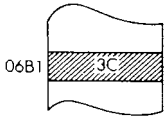
Dopo:





CODICE OGGETTO





INC (IX + d) Incrementa la locazione di memoria indirizzata in modo indicizzato (IX + d).

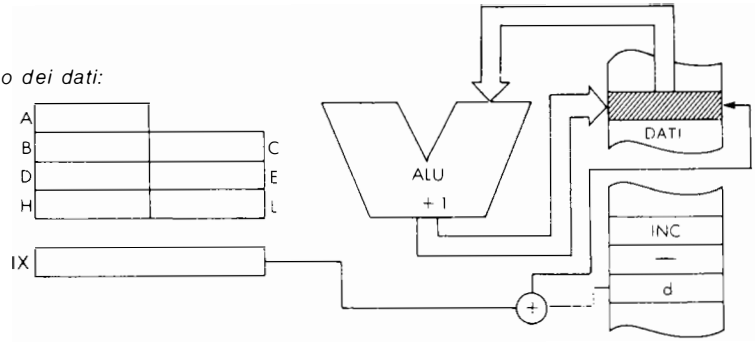
Funzione: (IX + d) ← (IX + d) + 1

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	1	0	1	0	0	byte 2: 34
← d →								byte 3: valore di offset

Descrizione: Il contenuto della locazione di memoria indirizzata dal contenuto di registri IX più il dato valore di offset è incrementato e immagazzinato di nuovo in quella posizione.

Flusso dei dati:



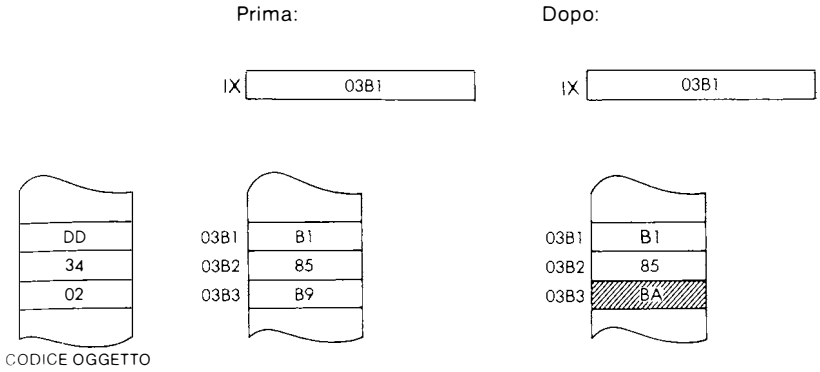
Temporizzazione: 6 cicli M; 23 stati T; 11.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indicizzato.

Flag:

S	Z		H		P/V	N	C
●	●		●		●	○	

Esempio: INC (IX + 2)



INC (IY + d)

Incrementa la locazione di memoria indirizzata in modo indicizzato (IY + d).

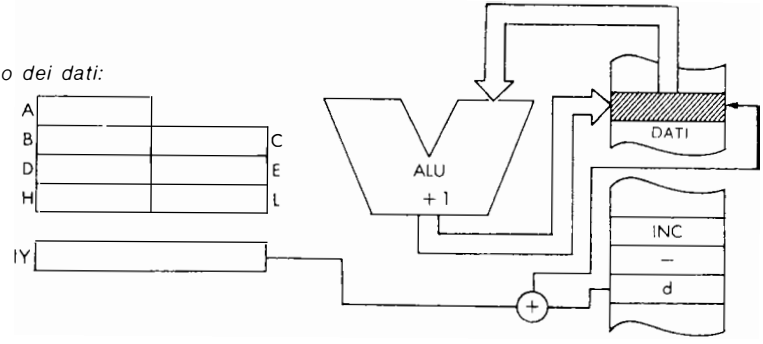
Funzione: (IY + d) ← (IY + d) + 1

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	1	0	1	0	0	byte 2: 34
d								byte 3: valore di offset

Descrizione: Il contenuto della locazione di memoria indirizzata dal contenuto del registro IY più il dato valore di offset è incrementato e immagazzinato di nuovo in quella posizione.

Flusso dei dati:



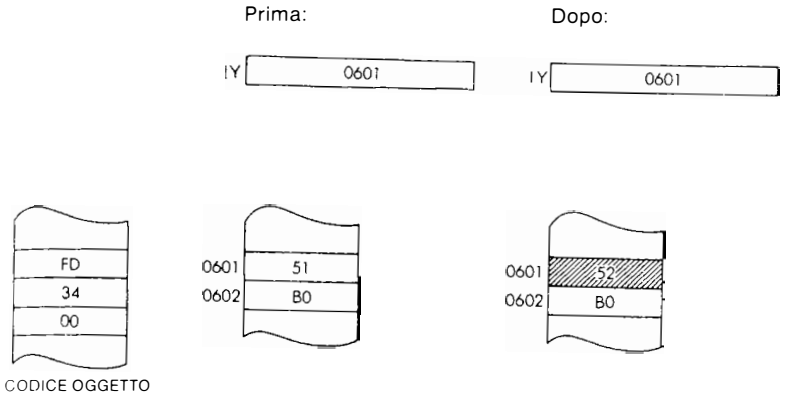
Temporizzazione: 6 cicli M; 23 stati T; 11.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indicizzato.

Flag:

S	Z		H		P/V	N	C
●	●		●		●	○	

Esempio: INC (IY + 0)



INC IX

Incrementa IX.

Funzione: IX ← IX + 1

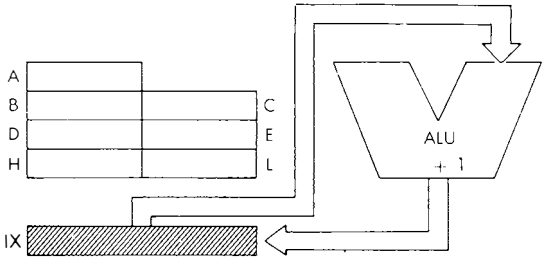
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	0	0	1	1	byte 2: 23

Descrizione:

Il contenuto del registro IX è incrementato e il risultato è immagazzinato di nuovo in IX.

Flusso dei dati:



Temporizzazione: 2 cicli M; 10 stati T; 5 μsec @ 2 MHz.

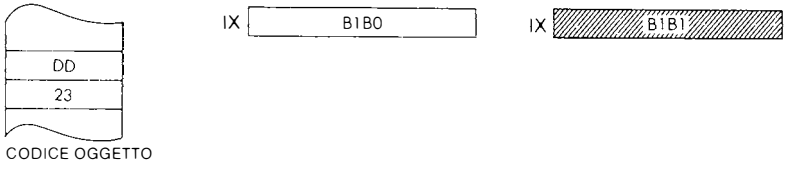
Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

Esempio: INC IX

Prima: Dopo:



INC IY

Incrementa IY.

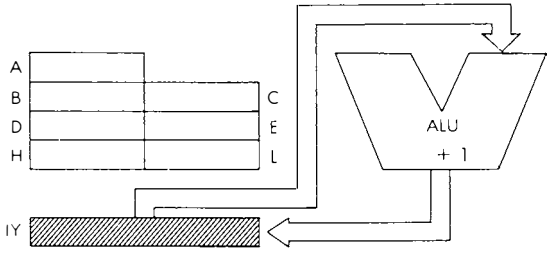
Funzione: IY ← IY + 1

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	0	0	1	1	byte 2: 23

Descrizione: Il contenuto del registro IY è incrementato e il risultato è immagazzinato di nuovo in IY.

Flusso dei dati:



Temporizzazione: 2 cicli M; 10 stati T; 5 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

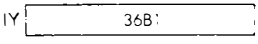
Flag:

S	Z	H	P/V	N	C

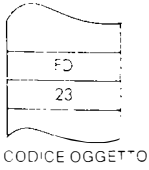
(nessun effetto)

Esempio: INC IY

Prima:



Dopo:



IND Ingresso con decremento.

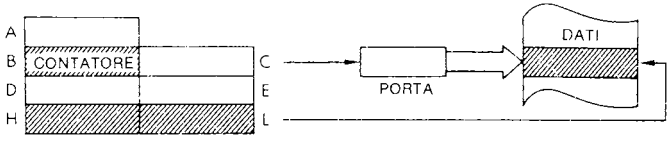
Funzione: (HL) ← (C); B ← B − 1; HL ← HL − 1.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	1	0	byte 2: AA

Descrizione: Il dispositivo periferico indirizzato dal registro C è letto ed il risultato è caricato nella locazione di memoria indirizzata dalla coppia di registri HL. Il registro B e la coppia di registri HL sono poi decrementati tutti e due.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T; 8 μsec @ 2 MHz.

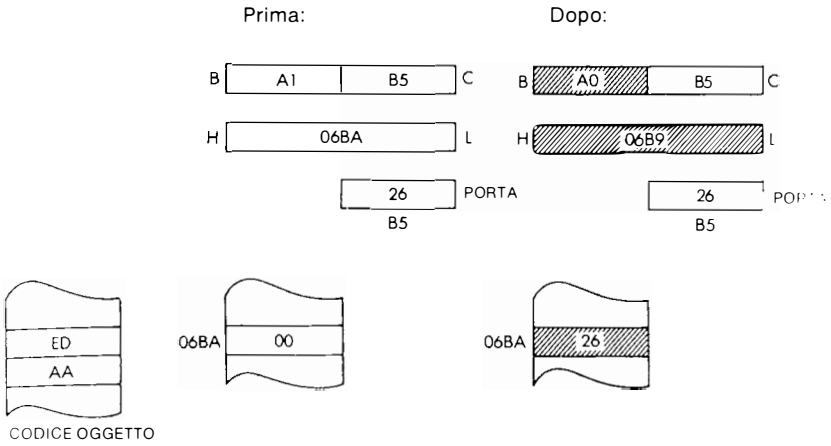
Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	X		?		?	1	

Posto ad uno se B = 0 dopo l'esecuzione. Altrimenti posto a 0.

Esempio: IND


Prima:
Dopo:
B
A1
B5
C
H
06BA
L
26
PORTA
B5
B
A0
B5
C
H
06B9
L
26
B5
ED
AA
CODICE OGGETTO
06BA
00
06BA
26

INDR

Ingresso di blocco e decremento.

Funzione:

(HL) ← (C); B ← B − 1; HL ← HL − 1.
Ripete fino a quando B = 0.

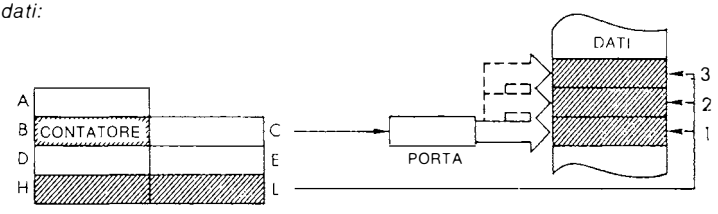
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	0	byte 2: BA

Descrizione:

Il dispositivo periferico indirizzato dal registro C è letto e il risultato è caricato nella locazione di memoria indirizzata della coppia di registri HL. Poi, il registro B e la coppia di registri HL sono decrementati. Se B non è zero, il PC è decrementato di due e l'istruzione è rieseguita.

Flusso dei dati:



Temporizzazione:

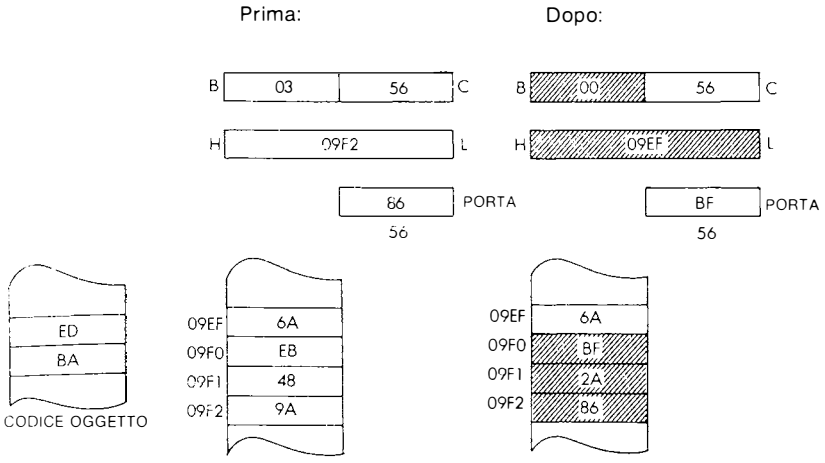
B = 0: 4 cicli M; 16 stati T; 8 μsec @ 2 MHz.
B ≠ 0: 5 cicli M; 21 stati T; 10.5 μsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	1		?		?	1	

Esempio: INDR



INI

Ingresso con incremento.

Funzione: (HL) ← (C); B ← B − 1; HL ← HL + 1

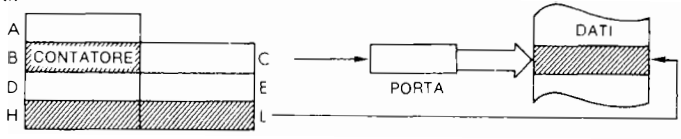
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	1	0	byte 2: A2

Descrizione: Il dispositivo periferico indirizzato, dal registro C è letto e il risultato è caricato nella locazione di memoria indirizzata dalla coppia di registri HL. Il registro B è decrementato ed è incrementata la coppia di registri HL.

Il contenuto di C è posto sulla metà inferiore del bus degli indirizzi. Il contenuto di B è posto sulla metà superiore. La selezione I/O è generalmente fatta da C, cioè, da A0 ad A7. B è un contatore di byte.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T; 8 µsec @ 2 MHz.

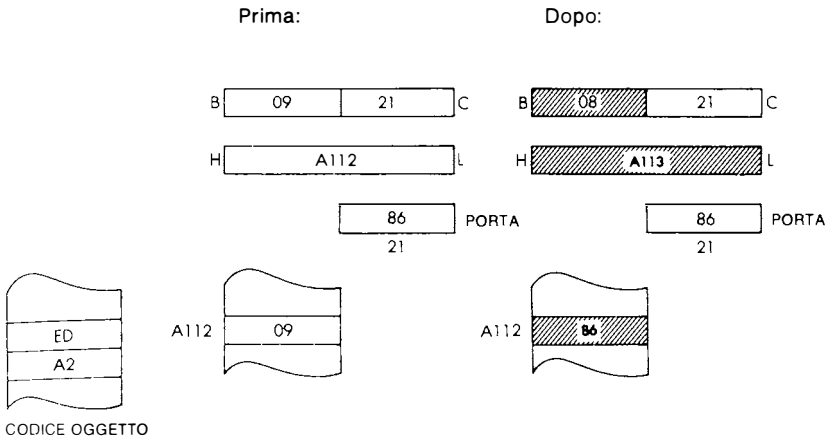
Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	X		?		?	1	

Z è posto ad uno se B = 0 dopo l'esecuzione.
Altrimenti è posto a 0.

Esempio: INI


Prima:
Dopo:
B
09
21
C
H
A112
L
86
PORTA
21
B
08
21
C
H
A113
L
86
PORTA
21
ED
A2
CODICE OGGETTO
A112
09
A112
86

INIR

Ingresso di blocco ed incremento.

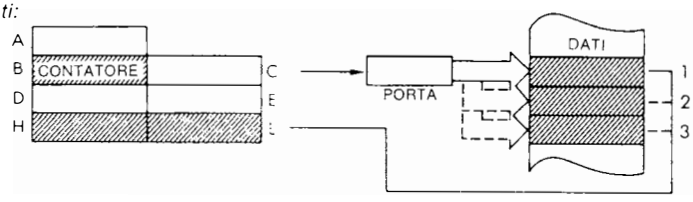
Funzione: (HL) ← (C); B ← B − 1; HL ← HL + 1; Ripete fino a quando B = 0.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	1	0	byte 2: B2

Descrizione: Il dispositivo periferico indirizzato dal registro C è letto e il risultato è caricato nella locazione di memoria indirizzata dalla coppia di registri HL. Il registro B è decrementato e la coppia di registri HL è incrementata. Se B non è zero, il PC è decrementato di 2 ed è rieseguita l'istruzione.

Flusso dei dati:



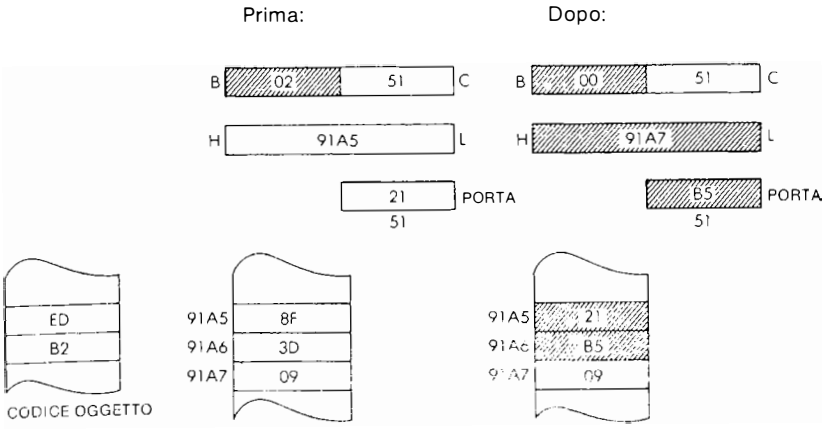
Temporizzazione: B = 0: 4 cicli M; 16 stati T; 8 µsec @ 2 MHz.
B ≠ 0: 5 cicli M; 21 stati T; 10.5 µsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	↕		?		?	1	

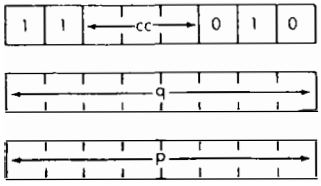
Esempio: INIR



JP cc, p q Salto condizionato alla locazione pq.

Funzione: se cc è vero: PC ← pq.

Formato:



byte 1

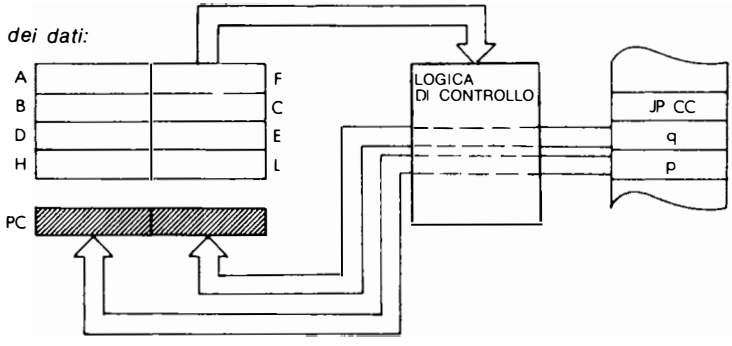
byte 2: indirizzo, ordine inferiore

byte 3: indirizzo, ordine superiore

Descrizione: Se la condizione specificata è vera, l'indirizzo di due byte che segue immediatamente il codice operativo sarà caricato nel PC con il primo byte che segue il codice operativo caricato nell'ordine inferiore del PC. L'indirizzo è ignorato se la condizione non è soddisfatta. cc può essere ognuno di questi:

NZ – 000	non zero
Z – 001	zero
NC – 010	nessun riporto
C – 011	riporto
PO – 100	parità dispari
PE – 101	parità pari
P – 110	più
M – 111	meno

Flusso dei dati:



Temporizzazione: 3 cicli M; 10 stati T; 5 μsec @ 2 MHz.

Modo d'indirizzamento: Immediato.

Codici del byte:

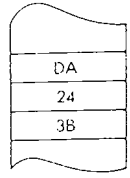
	NZ	Z	NC	C	PO	PE	P	M
C C	C2	CA	D2	DA	E2	EA	F2	FA

Flag:

S	Z		H		P/V	N	C

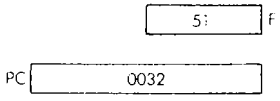
(nessun effetto)

Esempio: JP C, 3B24



CODICE OGGETTO

Prima:

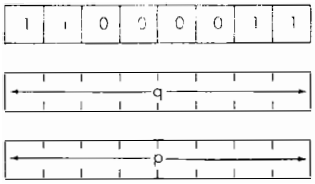


Dopo:

JP pq Saltare alla locazione pq.

Funzione: PC ← pq.

Formato:



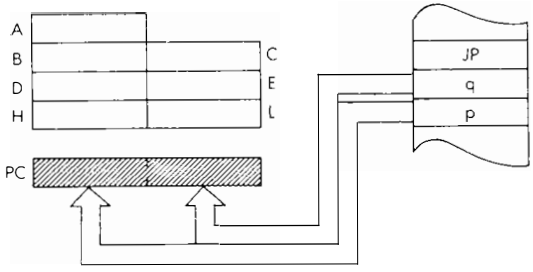
byte 1: C3

byte 2: indirizzo, ordine inferiore

byte 3: indirizzo, ordine superiore

Descrizione: Il contenuto della locazione di memoria che segue immediatamente il codice operativo è caricato nella metà di ordine inferiore del PC ed il contenuto della seconda locazione di memoria che segue immediatamente è caricato nell'ordine superiore del PC. L'istruzione successiva sarà prelevata da questo nuovo indirizzo.

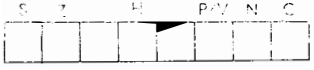
Flusso dei dati:



Temporizzazione: 3 cicli M; 10 stati T; 5 μsec @ 2 MHz.

Modo d'indirizzamento: Immediato.

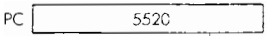
Flag:



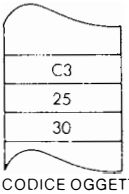
(nessun effetto)

Esempio: JP 3025

Prima:



Dopo:



CODICE OGGETTO

JP (HL)

Salta ad HL.

Funzione: PC ← HL

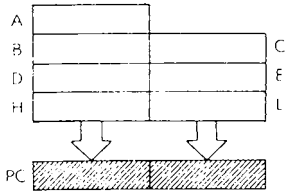
Formato:

1	1	1	0	1	0	0	

E9

Descrizione: Il contenuto della coppia di registri HL è caricato nel PC. L'istruzione successiva è prelevata a partire da questo nuovo indirizzo.

Flusso dei dati:



Temporizzazione: 1 cicli M; 4 stati T; 2 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

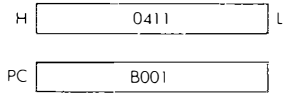
Flag:

S	Z		H		P/V	N	C

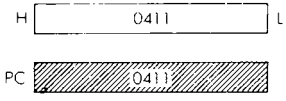
(nessun effetto)

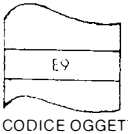
Esempio: JP (HL)

Prima:



Dopo:





CODICE OGGETTO

JP (IX)

Salta ad IX.

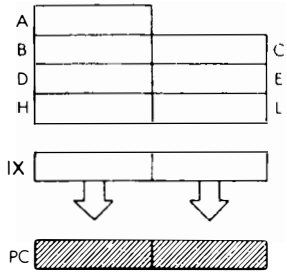
Funzione: PC ← IX

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	1	0	0	1	byte 2: E9

Descrizione: Il contenuto del registro IX è caricato nel PC. L'istruzione successiva è prelevata da questo nuovo indirizzo.

Flusso dei dati:



Temporizzazione: 2 cicli M; 8 stati T; 4 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

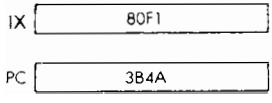
Flag:

S	Z		H		P/V	N	C

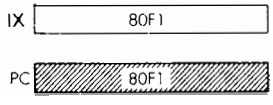
(nessun effetto)

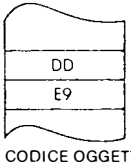
Esempio: JP (IX)

Prima:



Dopo:





CODICE OGGETTO

JP (IY)

Salta ad IY.

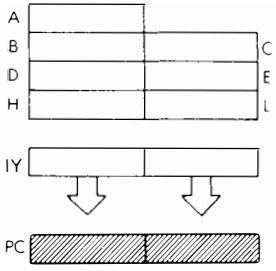
Funzione: PC ← IY

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	1	0	0	1	byte 2: E9

Descrizione: Il contenuto del registro IY è spostato nel PC. L'istruzione successiva sarà prelevata da questo nuovo indirizzo

Flusso dei dati:



Temporizzazione: 2 cicli M; 8 stati T; 4 µsec @ 2 MHz.

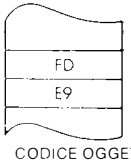
Modo d'indirizzamento: Implicito.

Flag:

S	Z	H	P/V	N	C

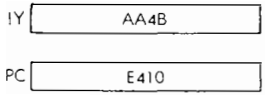
(nessun effetto)

Esempio: JP (IY)

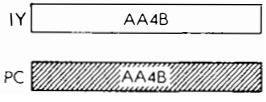


CODICE OGGETTO

Prima:



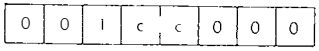
Dopo:



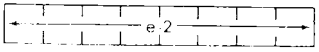
JR cc, e Salto condizionato di e in modo relativo.

Funzione: se cc è vero, PC ← PC + e

Formato:



byte 1

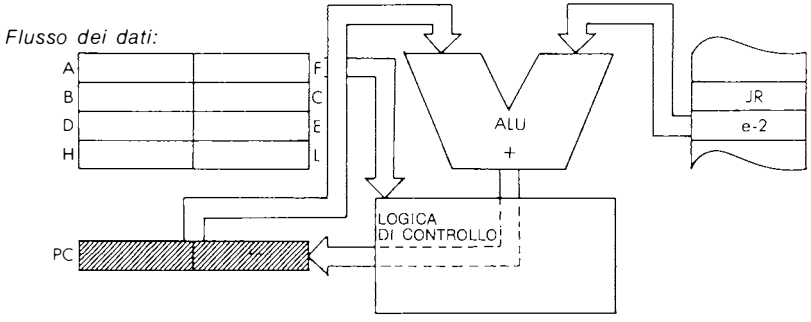


byte 2: valore di offset.

Descrizione: Se la condizione specificata è soddisfatta, il dato valore di offset è sommato al PC usando l'aritmetica in complemento a due così da permettere sia salti in avanti che indietro. Il valore di offset è sommato al valore di PC + 2 (dopo il salto). Come risultato, l'offset effettivo è da −126 a 129 byte. L'assemblatore sottrae automaticamente 2 dal valore di offset di sorgente per generare il codice esadecimale. Se la condizione non è soddisfatta, il valore di offset è ignorato e l'esecuzione delle istruzioni continua in sequenza. cc può essere ognuno di questi:

NZ — 00 NC — 10
Z — 01 C — 11

Flusso dei dati:



Temporizzazione:

	cicli M:	*stati T:*	*µsec @ 2 MHz:*
condizione soddisfatta:	3	12	6
condizione non soddisfatta:	2	7	3.5

Modo d'indirizzamento: Immediato.

Codici del byte:

cc.	NZ	Z	NC	C
	20	28	30	38

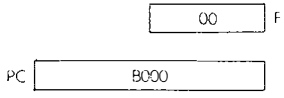
Flag:

S	Z		H		P/V	N	C

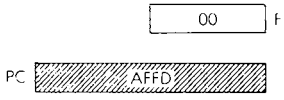
(nessun effetto)

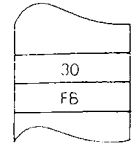
Esempio: JR NC, $ − 3 $ = PC corrente

Prima:



Dopo:





CODICE OGGETTO

JR e

Salto di e in modo relativo.

Funzione: PC ← PC + e

Formato:

0	0	0	1	1	0	0	0

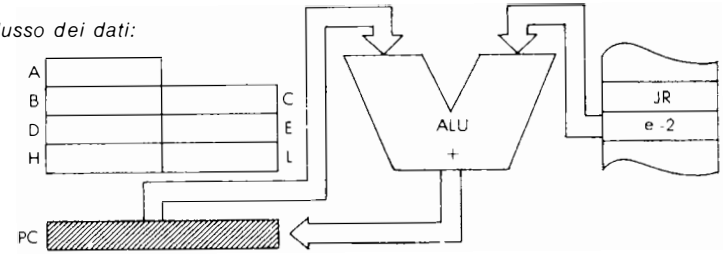
byte 1: 18

e-2

byte 2: valore di offset

Descrizione: Il dato valore di offset è sommato al PC usando l'aritmetica in complemento a due così da permettere salti sia in avanti che indietro. Il valore di offset è sommato al valore di PC + 2 (dopo il salto). Come risultato, l'offset effettivo è da −126 a +129 byte. L'assemblatore sottrae automaticamente 2 dal valore di offset di sorgente per generare il codice esadecimale.

Flusso dei dati:



Temporizzazione: 3 cicli M; 12 stati T; 6 µsec @ 2 MHz.

Modo d'indirizzamento: Immediato.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

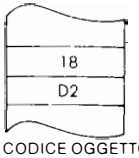
Esempio: JR DA

Prima:

PC B100

Dopo:

PC B0D4



CODICE OGGETTO

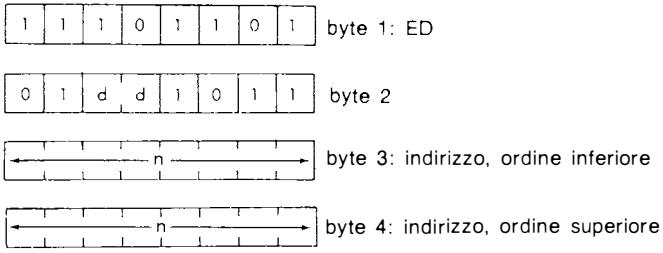
LD dd, (nn)

Carica la coppia di registri dd con le locazioni di memoria indirizzate da nn.

Funzione:

$dd_{basso} \leftarrow (nn)$; $dd_{alto} \leftarrow (nn + 1)$

Formato:

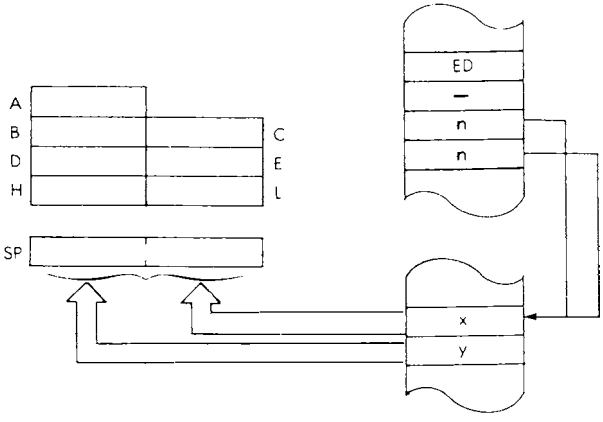


Descrizione:

Il contenuto della locazione di memoria indirizzata dalle locazioni di memoria che seguono immediatamente il codice operativo è caricato nell'ordine inferiore della coppia di registri specificata. Il contenuto della locazione di memoria che segue immediatamente quella precedentemente caricata è poi caricato nel byte di ordine superiore della coppia di registri. Il byte di ordine inferiore dell'indirizzo nn segue immediatamente il codice operativo. dd può essere ognuno di questi:

BC – 00	HL – 10
DE – 01	SP – 11

Flusso dei dati:



Temporizzazione. 6 cicli M; 20 stati T; 10 µsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

Codici del byte:

dd:	BC	DE	HL	SP
	4B	5B	6B	73

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

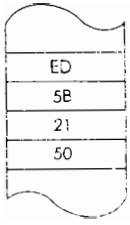
Esempio: LD DE, (5021)

Prima:

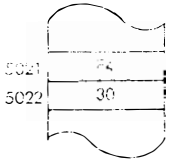
D | DBE2 | E

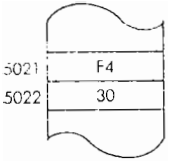
Dopo:

D | 30F4 | E



CODICE OGGETTO



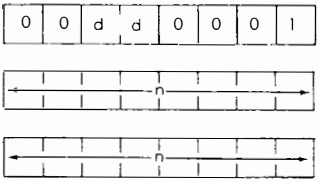


LD dd, nn

Carica la coppia di registri dd con il dato immediato nn.

Funzione: dd ← nn

Formato:



byte 1

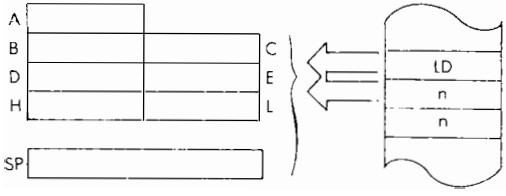
byte 2: dato immediato, ordine inferiore

byte 3: dato immediato, ordine superiore

Descrizione: Il contenuto delle due locazioni di memoria che seguono immediatamente il codice operativo è caricato nella coppia di registri specificata. Il byte di ordine inferiore del dato viene immediatamente dopo il codice operativo. dd può essere ognugno di questi

BC – 00 HL – 10
DE – 01 SP – 11

Flusso dei dati:



Temporizzazione: 3 cicli M; 10 stati T; 5 µsec @ 2 MHz.

Modo d'indirizzamento: Immediato.

Codice del byte:

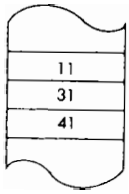
dd:	BC	DE	HL	SP
	01	11	21	31

Flag:

S	Z		H		P/V	N	C

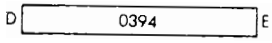
(nessun effetto)

Esempio: LD DE, 4131

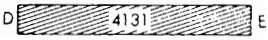


CODICE OGGETTO

Prima:



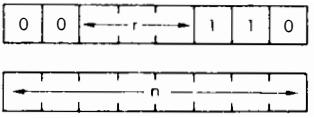
Dopo:



LD r, n Carica il registro r con il dato immediato n.

Funzione: r ← n

Formato:



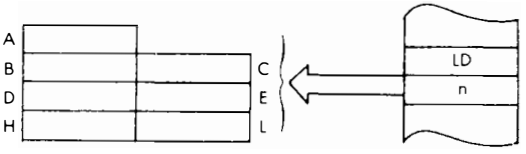
byte 1

byte 2: dato immediato

Descrizione: Il contenuto della locazione di memoria che segue immediatamente la posizione di codice operativo è caricato nel registro specificato. r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 µsec @ 2 MHz.

Modo d'indirizzamento: Immediato.

Codici del byte:

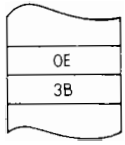
r:	A	B	C	D	E	H	L
	3E	06	0E	16	1E	26	2E

Flag:

S	Z		H		P/V	N	C

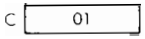
(nessun effetto)

Esempio: LD C, 3B



CODICE OGGETTO

Prima:



Dopo:

LD r, r' Carica il registro r dal registro r'.

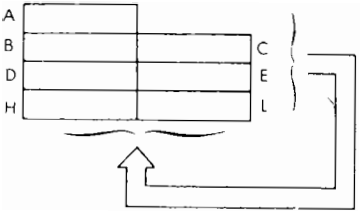
Funzione: r ← r'

Formato:

Descrizione: Il contenuto del registro di sorgente specificato è caricato nel registro di destinazione specificato. r e r' possono essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 1 cicli M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Codici del byte:

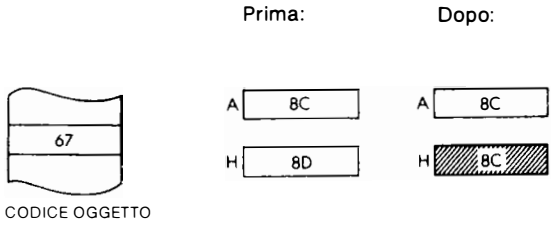
(dest.)	A	B	C	D	E	H	L (sorgente)
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	4F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD H, A



LD (BC), A

Carica la locazione di memoria indirettamente indirizzata (BC) dall'accumulatore.

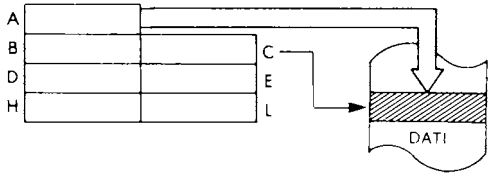
Funzione: (BC) ← A

Formato:

0	0	0	0	0	0	1	0	02

Descrizione: Il contenuto dell'accumulatore è caricato nella locazione di memoria indirizzata dal contenuto della coppia di registri BC.

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

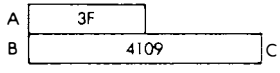
Flag:

S	Z	H	P/V	N	C

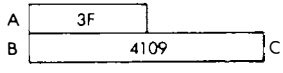
(nessun effetto)

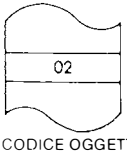
Esempio: LD (BC), A

Prima:

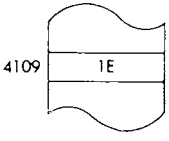


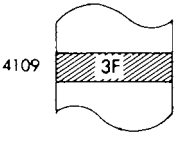
Dopo:





CODICE OGGETTO





LD (DE), A

Carica la locazione di memoria indirettamente indirizzata (DE) dall'accumulatore.

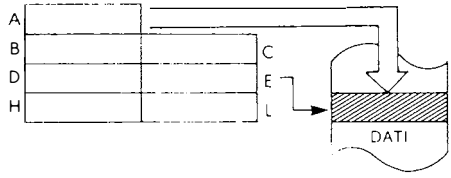
Funzione: (DE) ← A

Formato:

0	0	0	1	0	0	1	0	12

Descrizione: Il contenuto dell'accumulatore è caricato nella locazione di memoria indirizzata dal contenuto della coppia di registri DE.

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD (DE), A

Prima:

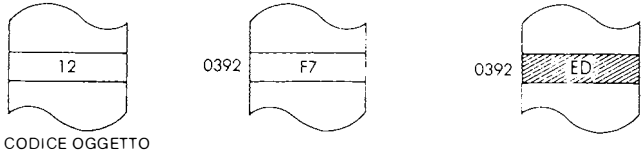
A [ED]

D [0392]

Dopo:

A [ED]

D [0392]



LD (HL), n

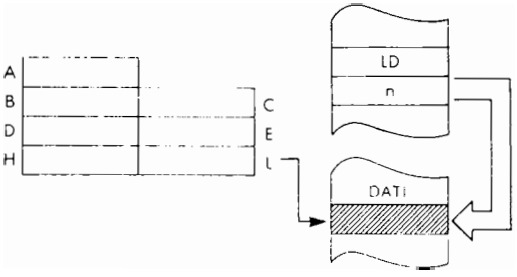
Carica il dato immediato n nella locazione di memoria indirizzata indirettamente (HL).

Funzione: (HL) ← n

Formato:

Descrizione Il contenuto della locazione di memoria che segue immediatamente il codice operativo è caricato nella locazione di memoria indirizzata indirettamente dal contatore dei dati HL.

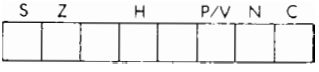
Flusso dei dati:



Temporizzazione: 3 cicli M; 10 stati T; 5 μsec @ 2 MHz.

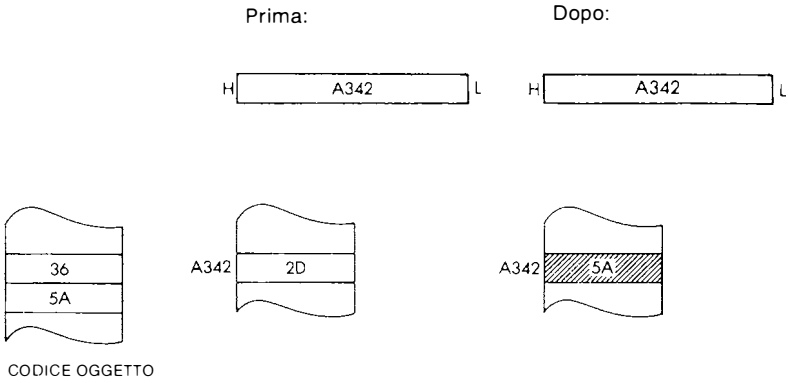
Modo d'indirizzamento: Immediato / indiretto.

Flag:



(nessun effetto)

Esempio: LD (HL), 5 A



LD (HL), r

Carica dal registro r la locazione di memoria indirizzata indirettamente da (HL).

Funzione: (HL) ← r

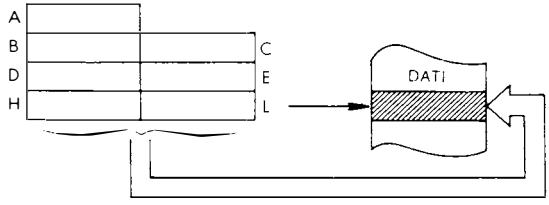
Formato:

0	1	1	1	0	← r →

Descrizione: Il contenuto del registro specificato è caricato nella locazione di memoria indirizzata dalla coppia di registri HL. r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Codici del byte:

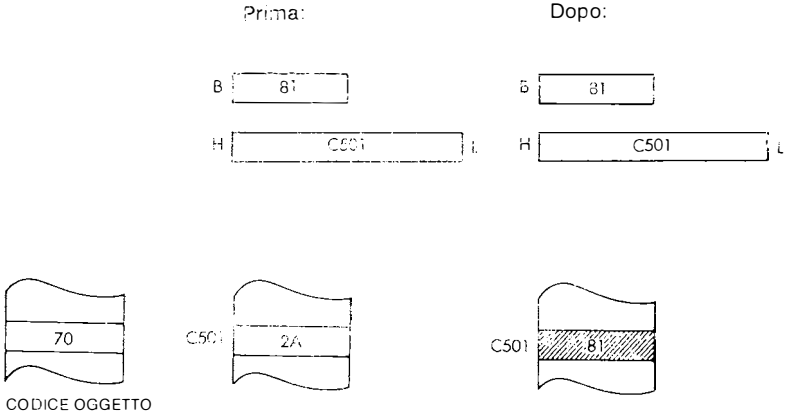
r:	A	B	C	D	E	H	L
	77	70	71	72	73	74	75

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD (HL), B

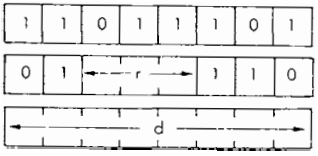


LD r, (IX + d)

Carica il registro indiretto r dalla locazione di memoria indicizzata (IX + d).

Funzione: r ← (IX + d)

Formato:



byte 1: DD

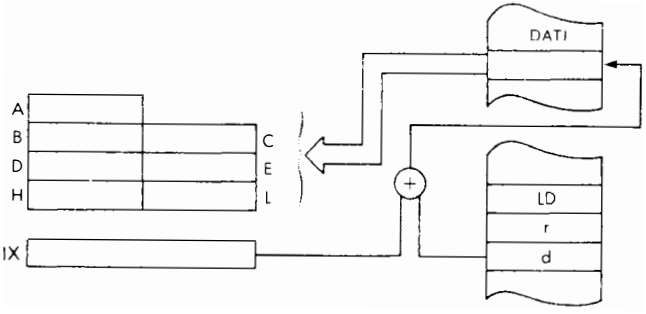
byte 2

byte 3: valore di offset

Descrizione: Il contenuto della locazione di memoria indirizzata dal registro indice IX più il dato valore di offset, è caricato nel registro specificato. r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indicizzato.

Codici del byte:

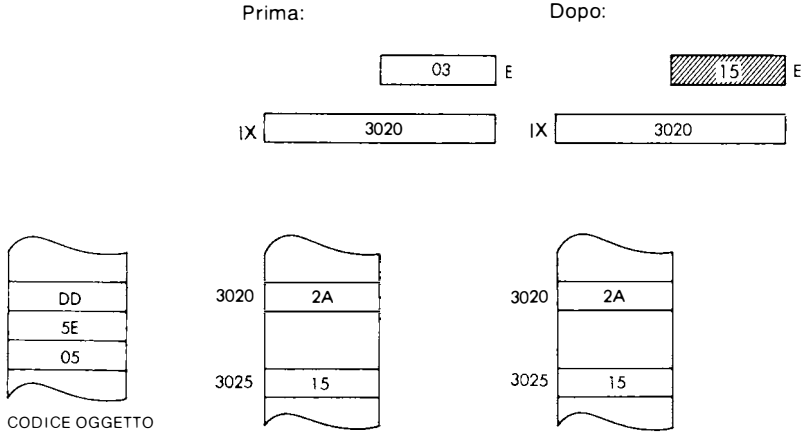
r:	A	B	C	D	E	H	L	
DD-	7E	46	4E	56	5E	66	6E	-d

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

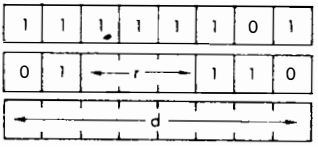
Esempio: LD E, (IX + 5)



LD r, (IY + d) Carica il registro indiretto r dalla locazione di memoria indirizzata in modo indicizzato (IY + d).

Funzione: r ← (IY + d)

Formato:



byte 1: FD

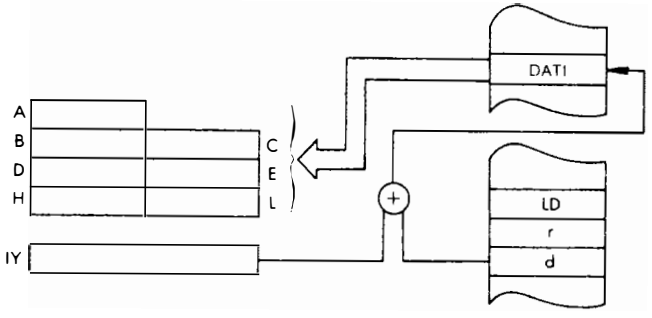
byte 2

byte 3: valore di offset

Descrizione: Il contenuto della locazione di memoria indirizzata dal registro di indice IY più il dato valore di offset, è caricato nel registro specificato. r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indicizzato.

Codici del byte:

r:	A	B	C	D	E	H	L	
FD -	7E	46	4E	56	56	66	6E	- d

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD A, (IY + 2)

Prima:

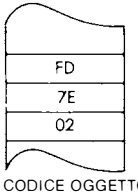
A [E3]

IY [B005]

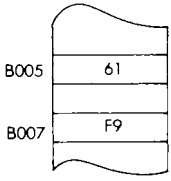
Dopo:

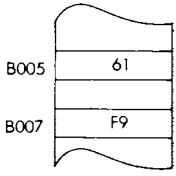
A [F9]

[B005]



CODICE OGGETTO

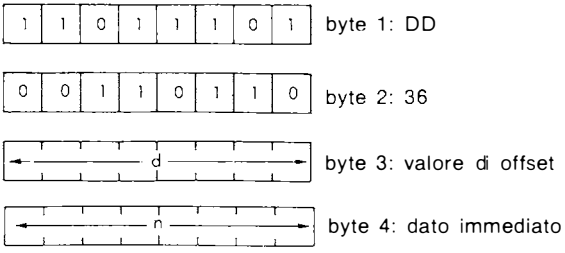




LD (IX + d), n Carica la locazione di memoria indirizzata in modo indicizzato (IX + d) con il dato immediato n.

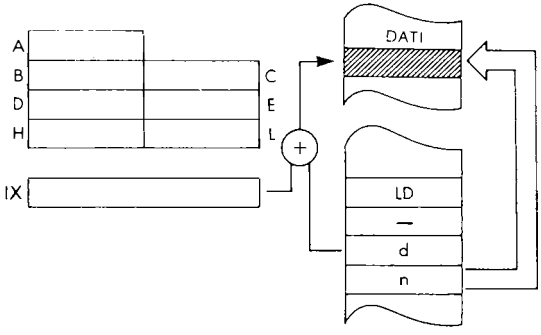
Funzione: (IX + d) ← n

Formato:



Descrizione: Il contenuto della locazione di memoria che segue immediatamente il codice operativo, è trasferito nella locazione di memoria indirizzata dal contenuto del registro di indice più il dato valore di offset.

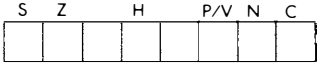
Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 µsec @ 2 MHz.

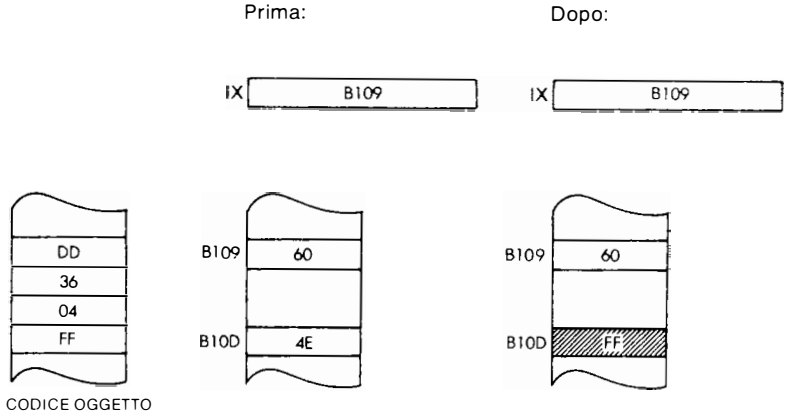
Modo d'indirizzamento: Indicizzato / immediato.

Flag:



(nessun effetto)

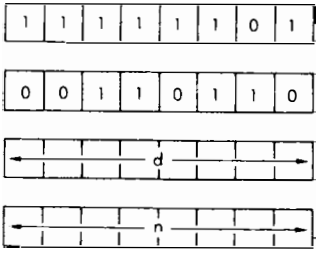
Esempio: LD (IX + 4), FF



LD (IY + d), n Carica la locazione di memoria indirizzata in modo indicizzato (IY + d) con il dato immediato n.

Funzione: (IY + d) ← n

Formato:



byte 1: FD

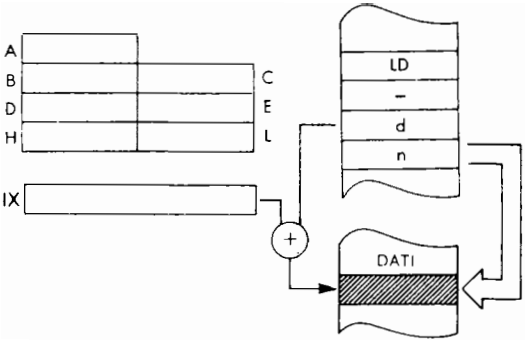
byte 2: 36

byte 3: valore di offset

byte 4: dato immediato

Descrizione: Il contenuto della locazione di memoria che segue immediatamente il codice operativo è trasferito nella locazione di memoria indirizzata dal contenuto del registro di indice più il dato valore di offset.

Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 µsec @ 2 MHz.

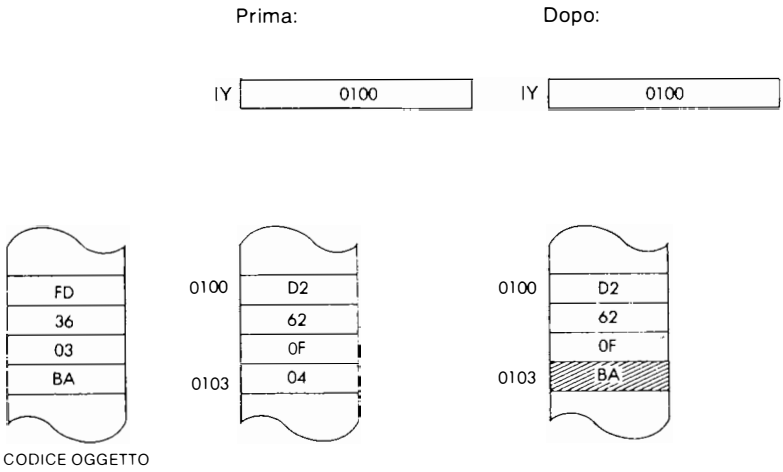
Modo d'indirizzamento: Indicizzato / immediato.

Flag:

S	Z		H		P/V	N	C

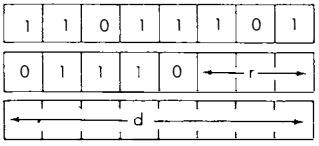
(nessun effetto)

Esempio: LD (IY + 3), BA



LD (IX + d), r Carica la locazione di memoria indirizzata in modo indicizzato (IX + d) dal registro r.

Formato:



byte 1: DD

byte 2

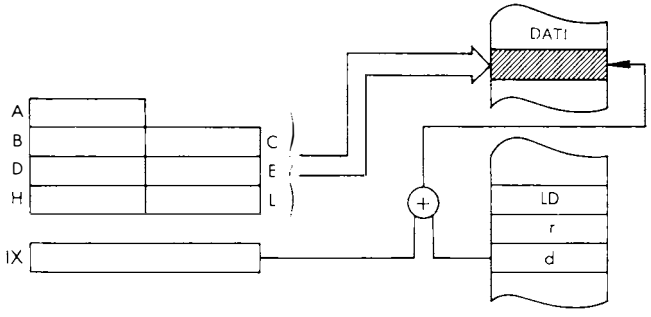
byte 3: valore di offset

Descrizione:

Il contenuto del registro specificato è caricato nella locazione di memoria indirizzata dal contenuto del registro di indice più il dato valore di offset.
r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indicizzato.

Codici del byte:

r:	A	B	C	D	E	H	L	
DD-	77	70	71	72	73	74	75	- d

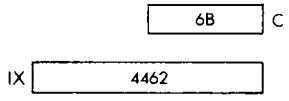
Flag:

S	Z		H		P/V	N	C

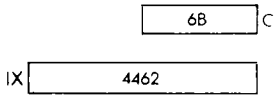
(nessun effetto)

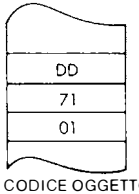
Esempio: LD (IX + 1), C

Prima:

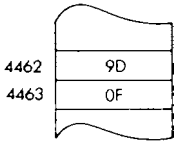


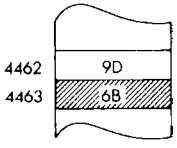
Dopo:





CODICE OGGETTO

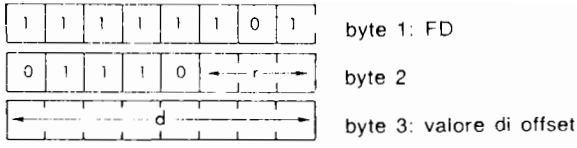




LD (IY + d), r Carica la locazione di memoria indirizzata in modo indicizzato (IY + d) dal registro r.

Funzione: (IY + d) ← r

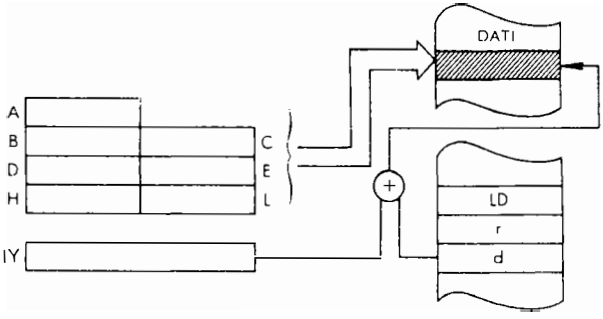
Formato:



Descrizione: Il contenuto del registro specificato è caricato nella locazione di memoria indirizzata dal contenuto del registro di indice più il dato valore di offset.
r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flusso dei dati:



Temporizzazione: 5 cicli M; 19 stati T; 9.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indicizzato.

Codice del byte:

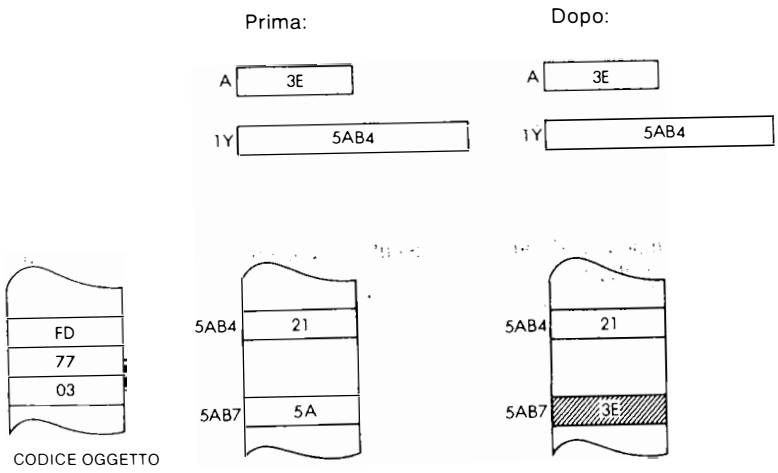
r:	A	B	C	D	E	H	L	
FD-	77	70	71	72	73	74	75	-d

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD (IY + 3), A

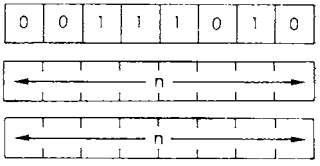


LD A, (nn)

Carica l'accumulatore dalla locazione di memoria (nn).

Funzione: A ← (nn)

Formato:



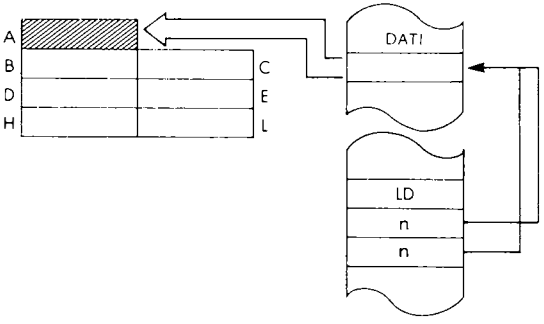
byte 1: 3A

byte 2: indirizzo, byte di ordine inferiore

byte 3: indirizzo, byte di ordine superiore

Descrizione Il contenuto della locazione di memoria indirizzata dal contenuto delle 2 locazioni di memoria che seguono immediatamente il codice operativo, è caricato nell'accumulatore. Il byte inferiore dell'indirizzo viene immediatamente dopo il codice operativo.

Flusso dei dati:



Temporizzazione: 4 cicli M; 13 stati T; 6.5 μsec @ 2 MHz.

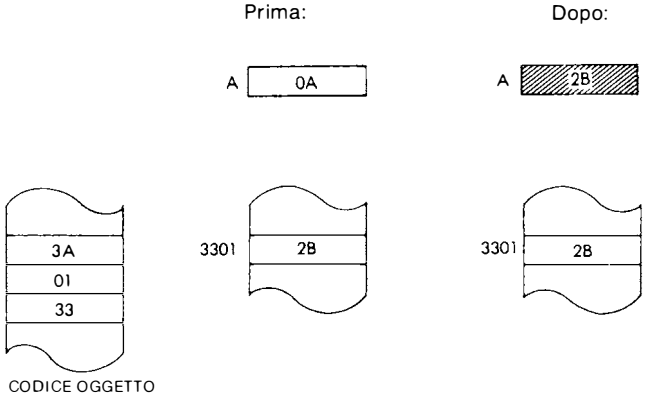
Modo d'indirizzamento: Diretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD A, (3301)



LD (nn), A

Carica la locazione di memoria indirizzata direttamente (nn) dall'accumulatore.

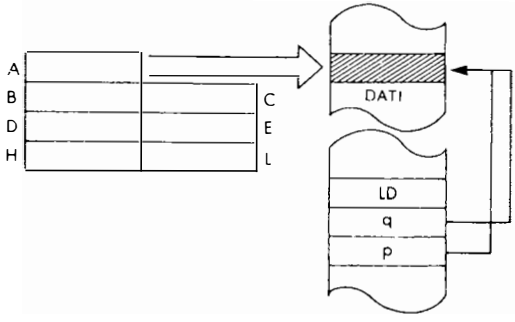
Funzione: (nn) ← A

Formato:

0	0	1	1	0	0	1	0	byte 1: 32
←			n				→	byte 2: indirizzo, ordine inferiore
←			n				→	byte 3: indirizzo, ordine superiore

Descrizione: Il contenuto dell'accumulatore è caricato nella locazione di memoria indirizzata dal contenuto delle posizioni di memoria che seguono immediatamente il codice operativo. Il byte inferiore dell'indirizzo segue immediatamente il codice operativo.

Flusso dei dati:



Temporizzazione: 4 cicli M; 13 stati T; 6.5 µsec @ 2 MHz.

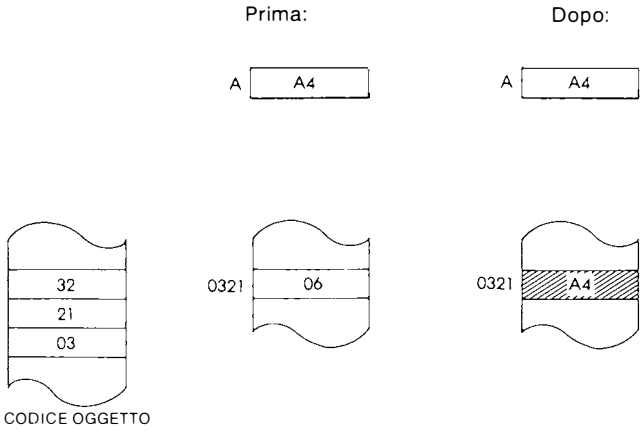
Modo d'indirizzamento: Diretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD (0321), A



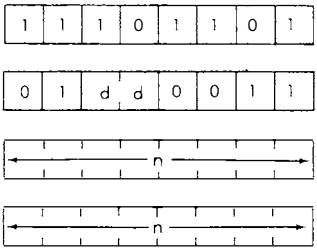
LD (nn), dd

Carica le locazioni di memoria indirizzate mediante nn dalla coppia di registri rr.

Funzione:

(nn) ← dd_{basso} ; (nn + 1) ← dd_{alto}

Formato:



byte 1: ED

byte 2

byte 3: indirizzo, ordine inferiore

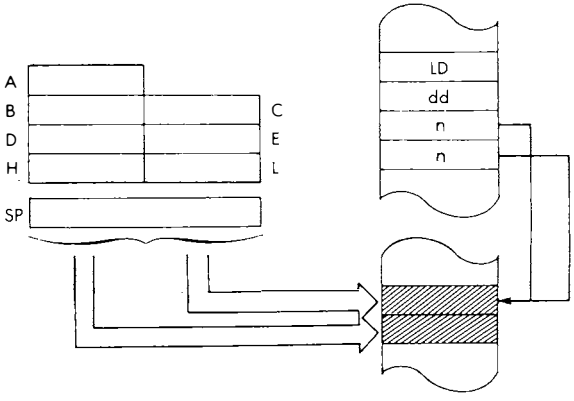
byte 4: indirizzo, ordine superiore

Descrizione:

Il contenuto del byte di ordine inferiore della coppia di registri specificata, è caricato nella locazione di memoria indirizzata dalle locazioni di memoria che seguono immediatamente il codice operativo. Il contenuto del byte di ordine superiore della coppia di registri, è caricato nella locazione di memoria che segue immediatamente quella caricata dall'ordine inferiore. L'ordine inferiore dell'indirizzo nn viene immediatamente dopo il codice operativo. dd può essere ognuno di questi:

BC – 00	HL – 10
DE – 01	SP – 11

Flusso dei dati:



Temporizzazione: 6 cicli M; 20 stati T; 10 µsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

Codici del byte:

dd:	BC	DE	HL	SP
ED-	43	53	63	73

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

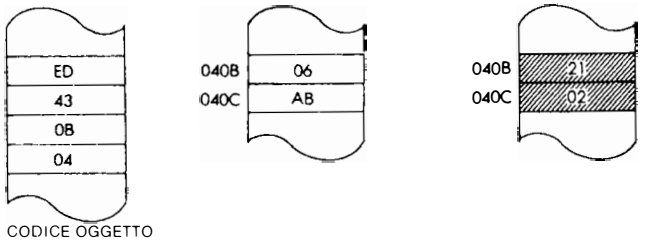
Esempio: LD (040 B), BC

Prima:

B [0221] C

Dopo:

B [0221] C

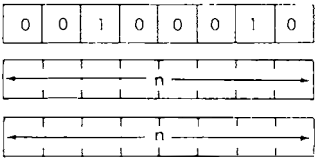


LD (nn), HL

Carica le locazioni di memoria indirizzate mediante nn da HL.

Funzione: (nn) ← L; (nn + 1) ← H

Formato:



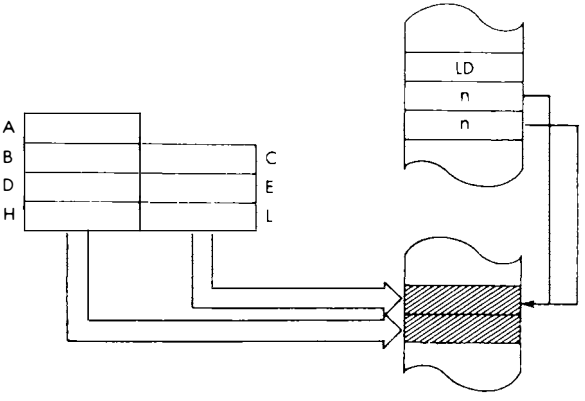
byte 1: 22

byte 2: indirizzo, ordine inferiore

byte 3: indirizzo, ordine superiore

Descrizione: Il contenuto del registro L è caricato nella locazione di memoria indirizzata dalle locazioni di memoria che seguono immediatamente il codice operativo. Il contenuto del registro H è caricato nella locazione di memoria che segue immediatamente la locazione caricata dal registro L. L'ordine inferiore dell'indirizzo nn avviene immediatamente dopo il codice operativo.

Flusso dei dati:



Temporizzazione: 5 cicli M; 16 stati T; 8 µsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

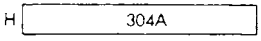
Flag:

S	Z		H		P/V	N	C

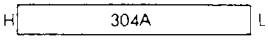
(nessun effetto)

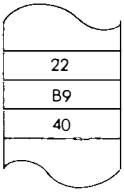
Esempio: LD (40B9), HL

Prima:

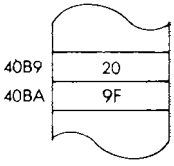


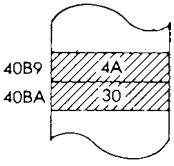
Dopo:





CODICE OGGETTO



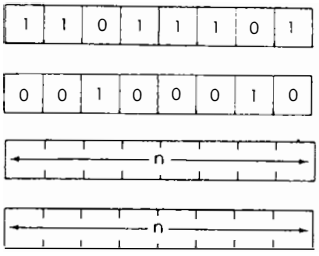


LD (nn), IX

Carica le locazioni di memoria indirizzate mediante nn dall'IX.

Funzione: (nn) ← IX_{basso} ; (nn + 1) ← IX_{alto}

Formato:



byte 1: DD

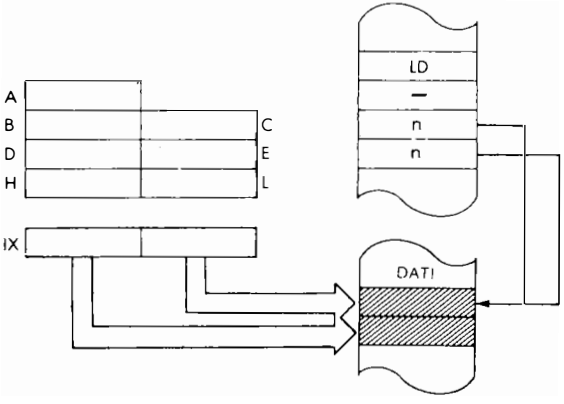
byte 2: 22

byte 3: indirizzo, ordine inferiore

byte 4: indirizzo, ordine superiore

Descrizione: Il contenuto di ordine inferiore del registro IX è caricato nella locazione di memoria indirizzata dal contenuto della locazione di memoria che segue immediatamente il codice operativo. Il contenuto di ordine superiore del registro IX è caricato nella locazione di memoria che segue immediatamente quello caricato dall'ordine inferiore. L'ordine inferiore dell'indirizzo nn avviene immediatamente dopo il codice operativo.

Flusso dei dati:



Temporizzazione: 6 cicli M; 20 stati T; 10 µsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

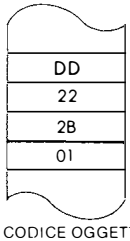
Esempio: LD (012B), IX

Prima:

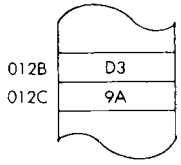
IX 0406

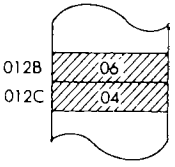
Dopo:

IX 0406



CODICE OGGETTO



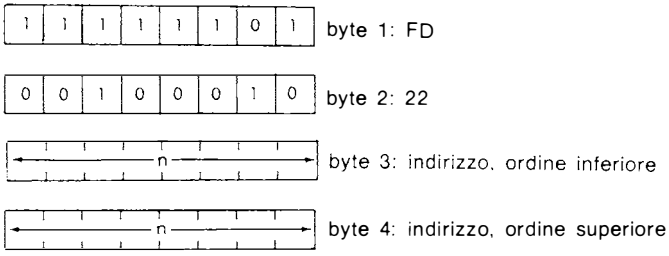


LD (nn), IY

Carica le locazioni di memoria indirizzate da nn dall'IY.

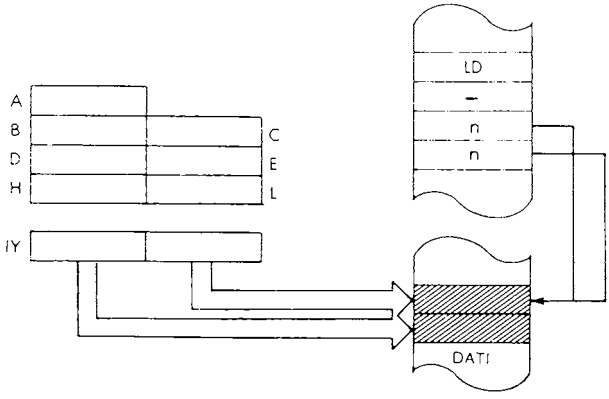
Funzione: (nn) ← IY_{basso} ; (nn + 1) ← IY_{alto}

Formato:



Descrizione: Il contenuto di ordine inferiore del registro IY è caricato nella locazione di memoria indirizzata dal contenuto delle locazioni di memoria che seguono immediatamente il codice operativo. Il contenuto di ordine superiore del registro IY è caricato nella locazione di memoria che segue immediatamente quello caricato dall'ordine inferiore. L'ordine inferiore dell'indirizzo nn viene immediatamente dopo il codice operativo.

Flusso dei dati.



Temporizzazione: 6 cicli M; 20 stati T; 10 μsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

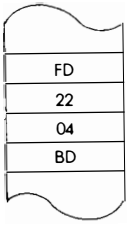
Esempio: LD (BD04), IY

Prima:

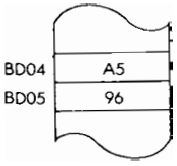
IY D204

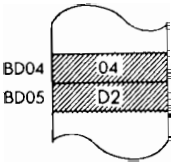
Dopo:

IY D204



CODICE OGGETTO





LD A, (BC)

Carica l'accumulatore dalla locazione di memoria indirizzata indirettamente mediante coppia di registri BC.

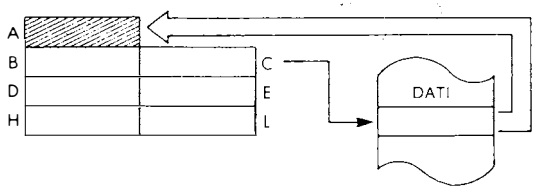
Funzione: A ← (BC)

Formato:

0	0	0	0	1	0	1	0	0A

Descrizione: Il contenuto della locazione di memoria indirizzata dal contenuto della coppia di registri BC, è caricato nell'accumulatore.

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

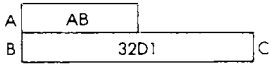
Flag:

S	Z		H		P/V	N	C

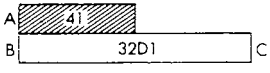
(nessun effetto)

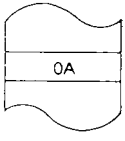
Esempio: LD A, (BC)

Prima:

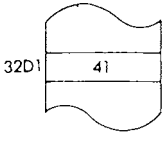


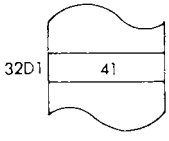
Dopo:





CODICE OGGETTO





LD A, (DE)

Carica l'accumulatore dalla locazione di memoria indirizzata indirettamente dalla coppia di registri DE.

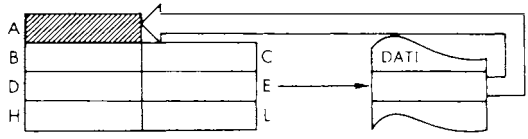
Funzione: A ← (DE)

Formato:

0	0	0	1	1	0	1	0	1A

Descrizione: Il contenuto della locazione di memoria indirizzata dal contenuto della coppia di registri DE, è caricato nell'accumulatore.

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD A, (DE)

Prima:

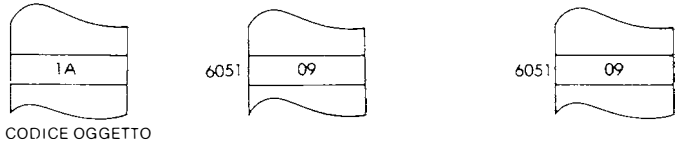
A [D2]

D [6051]

Dopo:

A [09]

D [6051] E



CODICE OGGETTO

LD A, I

Carica l'accumulatore dal registro del vettore d'interrupt I.

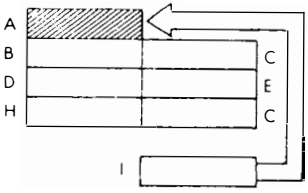
Funzione: A ← I

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	0	1	1	1	byte 2: 57

Descrizione: Il contenuto del registro del vettore d'interrupt è caricato nell'accumulatore.

Flusso dei dati:



Temporizzazione: 2 cicli M; 9 stati T; 4.5 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

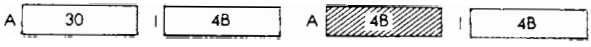
Flag:

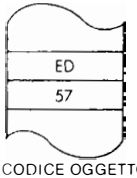
S	Z	H	P/V	N	C
●	●		○	×	○	

Posiziona il contenuto di IFF2.

Esempio: LD A, I

Prima: Dopo:





CODICE OGGETTO

LD I, A

Carica il registro del Vettore d'Interrupt I dall'accumulatore.

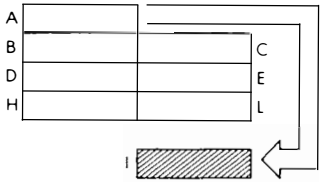
Funzione: I ← A

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	1	1	byte 2: 47

Descrizione: Il contenuto dell'accumulatore è caricato nel registro del Vettore d'Interrupt.

Flusso dei dati:



Temporizzazione: 2 cicli M; 9 stati T; 4.5 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

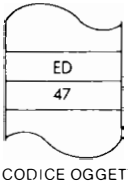
Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

Esempio: LD I, A

Prima: A 06 I D2

Dopo: A 06 I 06



CODICE OGGETTO

LD A, R

Carica l'accumulatore dal registro di Rinfresco della Memoria R.

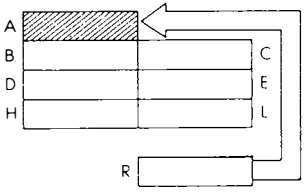
Funzione: A ← R

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	1	1	1	1	byte 2: 5F

Descrizione: Il contenuto del registro di Rinfresco della Memoria è caricato nell'accumulatore.

Flusso dei dati:



Temporizzazione: 2 cicli M; 9 stati T; 4.5 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

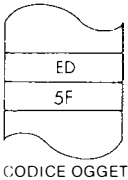
S	Z		H		P/V	N	C
●	●		○		×	○	

Posiziona il contenuto di IFF 2.

Esempio: LD A, R

Prima: A 62 R 4A

Dopo: A 4A R 4A



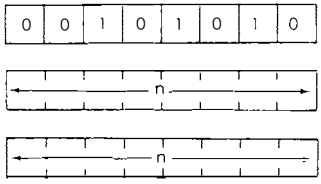
CODICE OGGETTO

LD HL, (nn)

Carica il registro HL dalle locazioni di memoria indirizzate da nn.

Funzione: L ← (nn); H ← (nn + 1)

Formato:



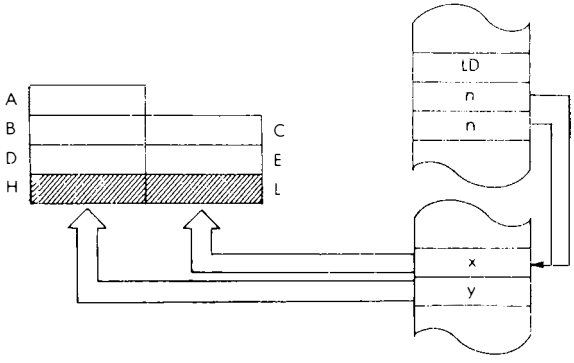
byte 1: 2A

byte 2: indirizzo, ordine inferiore

byte 3: indirizzo, ordine superiore

Descrizione: Il contenuto della locazione di memoria indirizzata dalle locazioni di memoria immediatamente dopo il codice operativo, è caricato nel registro L. Il contenuto della locazione di memoria dopo quella caricata nel registro L, è caricato nel registro H. Il byte inferiore dell'indirizzo avviene immediatamente dopo il codice operativo.

Flusso dei dati:



Temporizzazione: 5 cicli M; 16 stati T; 8 μsec @ 2 MHz.

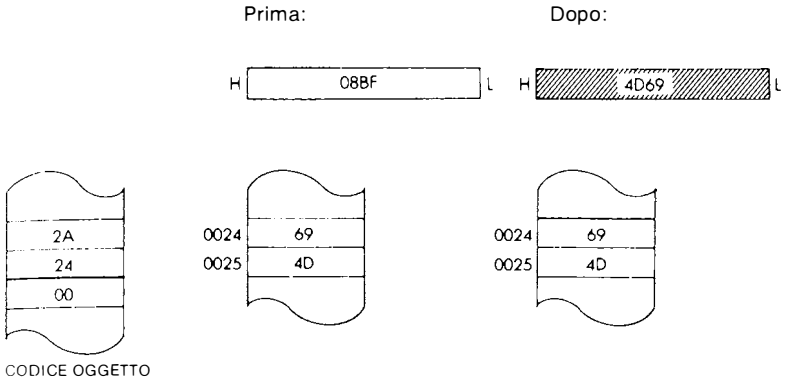
Modo d'indirizzamento: Diretto

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD HL, (0024)

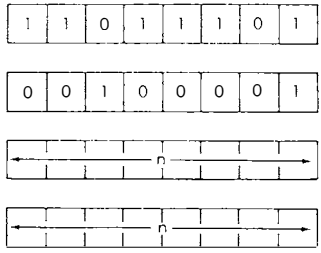


LD IX, nn

Carica il registro IX con il dato immediato nn.

Funzione: IX ← nn

Formato:



byte 1: DD

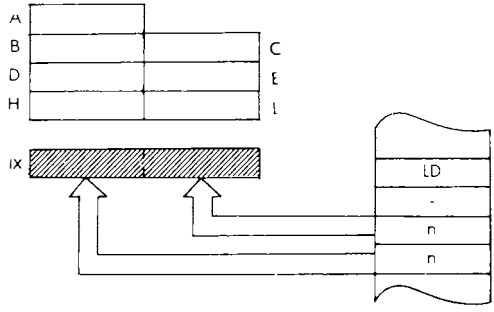
byte 2: 21

byte 3: dato immediato, ordine inferiore

byte 4: dato immediato, ordine superiore

Descrizione: Il contenuto delle locazioni di memoria che seguono immediatamente il codice operativo è caricato nel registro IX. Il byte d'ordine inferiore avviene immediatamente dopo il codice operativo.

Flusso dei dati:



Temporizzazione: 4 cicli M; 14 stati T; 7 μsec @ 2 MHz.

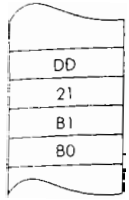
Modo d'indirizzamento: Immediato.

Flag:

S	Z		H		P/V	N	C

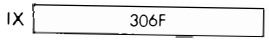
(nessun effetto)

Esempio: LD IX, BOB1



CODICE OGGETTO

Prima:

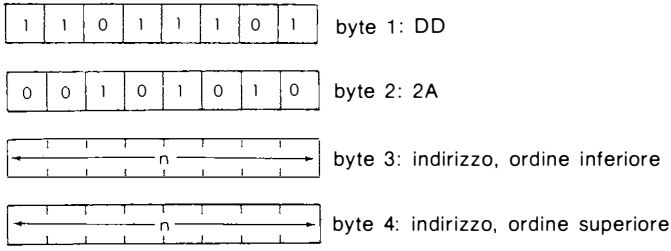


Dopo:

LD IX, nn Carica il registro IX dalle locazioni di memoria indirizzate da nn.

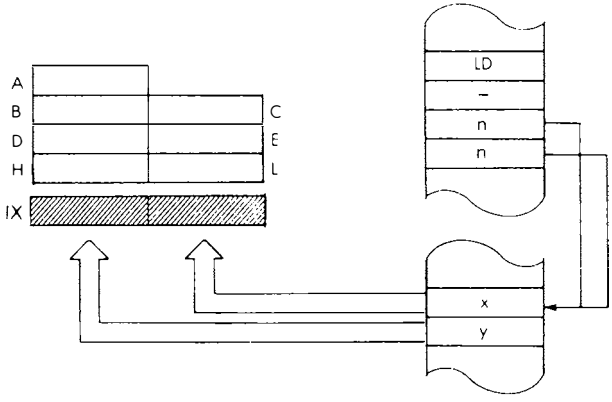
Funzione: $IX_{basso} \leftarrow (nn)$; $IX_{alto} \leftarrow (nn + 1)$

Formato:



Descrizione: Il contenuto della locazione di memoria indirizzata dalle locazioni di memoria che seguono immediatamente il codice operativo, è caricato nell'ordine inferiore del registro IX. Il contenuto della locazione di memoria che segue immediatamente quella caricata nell'ordine inferiore, è caricato nell'ordine superiore del registro IX. L'ordine inferiore dell'indirizzo nn segue immediatamente il codice operativo.

Flusso dei dati:



Temporizzazione: 6 cicli M; 20 stati T; 10 µsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

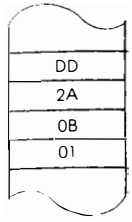
Esempio: LD IX, (010B)

Prima:

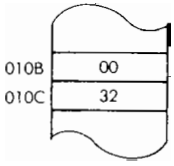
IX FF4B

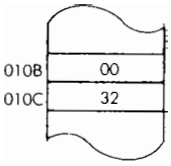
Dopo:

IX 3200



CODICE OGGETTO



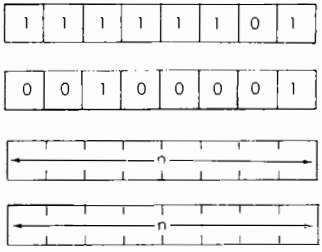


LD IY, (nn)

Carica il registro IY con il dato immediato nn.

Funzione: IY ← nn

Formato:



byte 1: FD

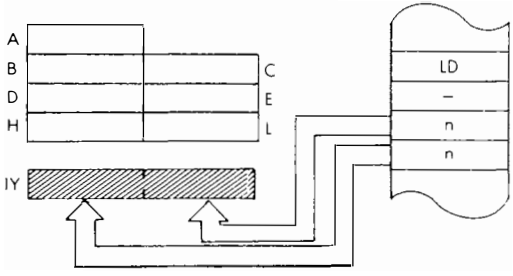
byte 2: 21

byte 3: dato immediato, ordine inferiore

byte 4: dato immediato, ordine superiore

Descrizione: Il contenuto delle locazioni di memoria che seguono immediatamente il codice operativo, è caricato nel registro IY. Il byte di ordine inferiore viene immediatamente dopo il codice operativo.

Flusso dei dati:



Temporizzazione: 4 cicli M; 14 stati T; 7 µsec @ 2 MHz.

Modo d'indirizzamento: Immediato.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

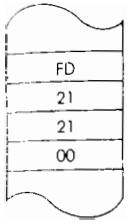
Esempio: LD IY, 21

Prima:

IY 069B

Dopo:

IY 0021



CODICE OGGETTO

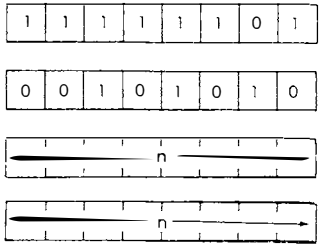
LD IY, nn

Carica il registro IY dalle locazioni di memoria indirizzate da nn.

Funzione:

$IY_{basso} \leftarrow (nn)$; $IY_{alto} \leftarrow (nn + 1)$

Formato:



byte 1: FD

byte 2: 2A

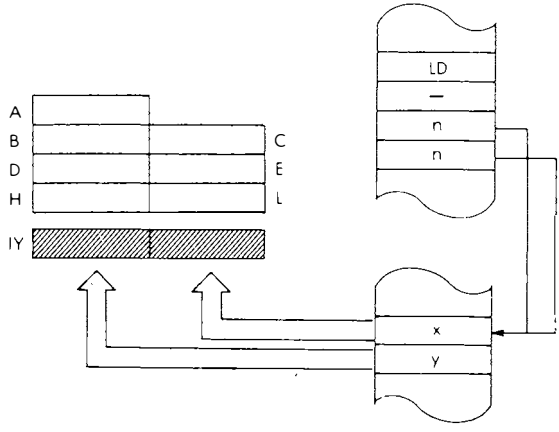
byte 3: indirizzo, ordine inferiore

byte 4: indirizzo, ordine superiore

Descrizione:

Il contenuto della locazione di memoria indirizzata dalle locazioni di memoria che seguono immediatamente il codice operativo, è caricato nell'ordine inferiore del registro IY. Il contenuto della locazione di memoria che segue immediatamente quella caricata nell'ordine inferiore, è caricato nell'ordine superiore del registro IY. L'ordine inferiore dell'indirizzo nn segue immediatamente il codice operativo.

Flusso dei dati:



Temporizzazione: 6 cicli M; 20 stati T; 10 µsec @ 2 MHz.

Modo d'indirizzamento: Diretto.

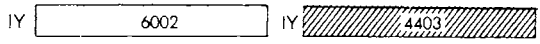
Flag:

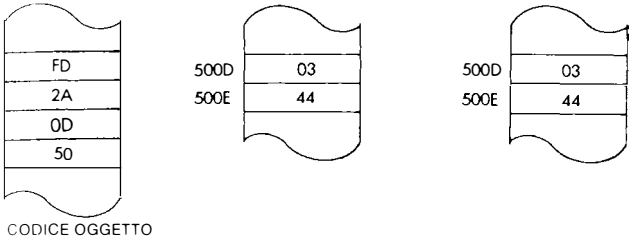
S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD IY, (500 D)

Prima: Dopo:





CODICE OGGETTO

LD R, A Carica il registro di Rinfresco di Memoria R dall'accumulatore.

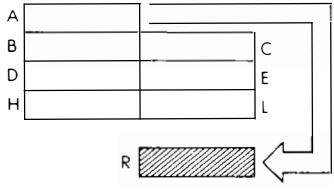
Funzione: R ← A

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	1	1	byte 2: 4F

Descrizione: Il contenuto dell'accumulatore è caricato nel registro Rinfresco di Memoria.

Flusso dei dati:



Temporizzazione: 2 cicli M; 9 stati T; 4.5 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

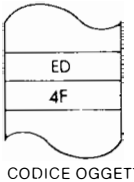
Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: LD R, A

Prima: Dopo:



CODICE OGGETTO

LD SP, HL

Carica il puntatore dello stack da HL.

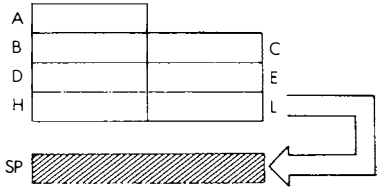
Funzione: SP ← HL.

Formato:

1	1	1	1	1	0	0	1	F9

Descrizione: Il contenuto della coppia di registri HL è caricato nel puntatore di stack.

Flusso dei dati:



Temporizzazione: 1 ciclo M; 6 stati T; 3 µsec @ 2 MHz.

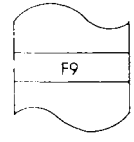
Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

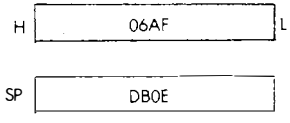
(nessun effetto)

Esempio: LD SP, HL

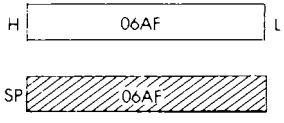


CODICE OGGETTO

Prima:



Dopo:



LD SP, IX

Carica il puntatore dello stack dal registro IX.

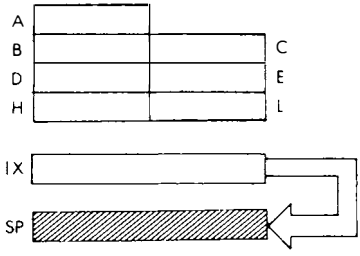
Funzione: SP ← IX

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	1	1	0	0	1	byte 2: F9

Descrizione: Il contenuto del registro IX è caricato nel puntatore dello stack.

Flusso dei dati:



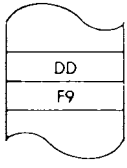
Temporizzazione: 2 cicli M; 10 stati T; 5 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

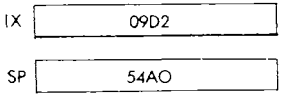
S	Z		H		P/V	N	C	
								(nessun effetto)

Esempio: LD SP, IX

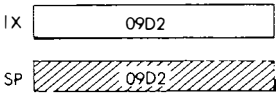


CODICE OGGETTO

Prima:



Dopo:



LD SP, IY

Carica il puntatore dello stack dal registro IY.

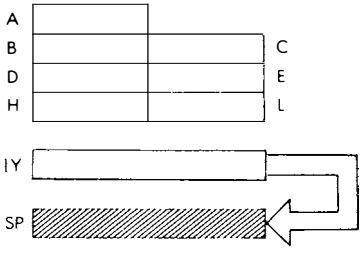
Funzione: SP ← IY

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	1	1	0	0	1	byte 2: F9

Descrizione: Il contenuto del registro IY è caricato nel puntatore dello stack.

Flusso dei dati:



Temporizzazione: 2 cicli M; 10 stati T; 5 µsec @ 2 MHz.

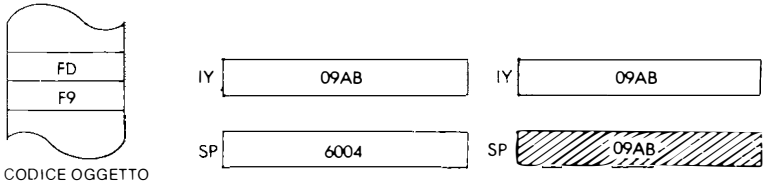
Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

Esempio: LD SP, IY

Prima: Dopo:



LDD

Carica il blocco e decrementa.

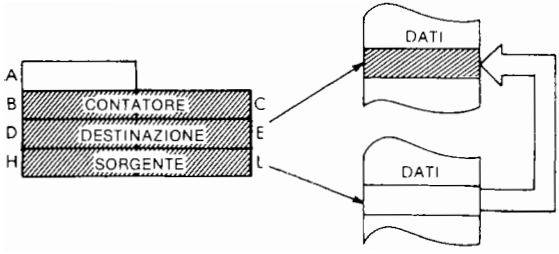
Funzione: (DE) ← (HL); DE ← DE − 1; HL ← HL − 1; BC ← BC − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	0	0	byte 2: A8

Descrizione: Il contenuto della locazione di memoria indirizzata da HL, è caricato nella locazione di memoria indirizzata da DE. Poi BC, DE, e HL sono tutti decrementati.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T; 8 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C
			O		X	O	

Posto a 0 se BC = 0 dopo l'esecuzione, altrimenti posto ad 1.

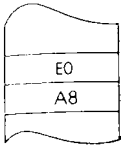
Esempio: LDD

Prima:

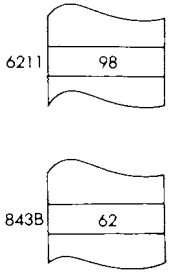
B	0B04	C
D	6211	C
H	843B	L

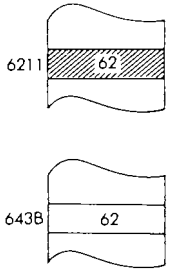
Dopo:

B	0B03	C
D	6210	E
H	843A	L



CODICE OGGETTO





LDDR

Ripeti il caricamento del blocco e decrementa.

Funzione: (DE) ← (HL); DE ← DE − 1; HL ← HL − 1; BC ← BC − 1; Ripeti fino a quando BC = 0.

Formato:

1	1	1	0	1	1	0	1

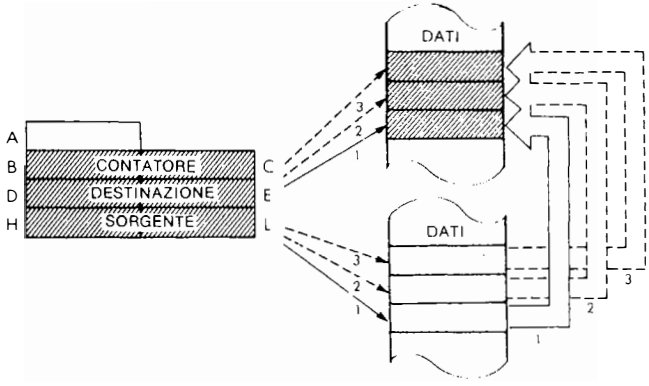
byte 1: ED

1	0	1	1	1	0	0	0

byte 2: B8

Descrizione: Il contenuto della locazione di memoria indirizzata dall'HL, è caricato nella posizione di memoria indirizzata da DE. Poi DE, HL, e BC sono tutti decrementati. Se BC ≠ 0, allora il PC è decrementato di 2 ed è rieseguita l'istruzione.

Flusso dei dati:



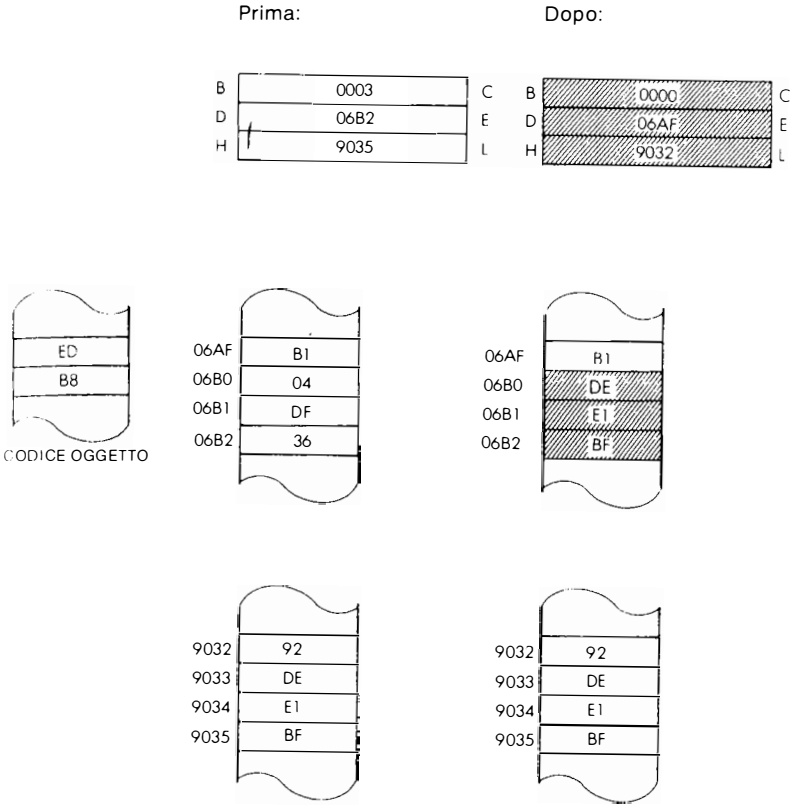
Temporizzazione: BC ≠ 0: 5 cicli M; 21 stati T; 10.5 μsec @ 2 MHz.
BC = 0: 4 cicli M; 16 stati T; 8 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C
			○		○	○	

Esempio: LDDR



LDI Carica il blocco ed incrementa.

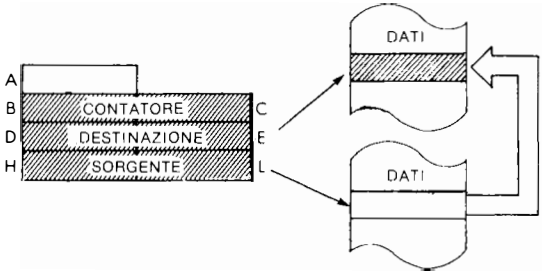
Funzione: (DE) ← (HL); DE ← DE + 1; HL ← HL + 1; BC ← BC − 1.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	0	0	byte 2: AO

Descrizione: Il contenuto della locazione di memoria indirizzata da HL, è caricato nella locazione di memoria indirizzata da DE. Poi, sia DE che HL sono incrementati, ed è decrementata la coppia di registri BC.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T; 8 µsec @ 2 MHz.

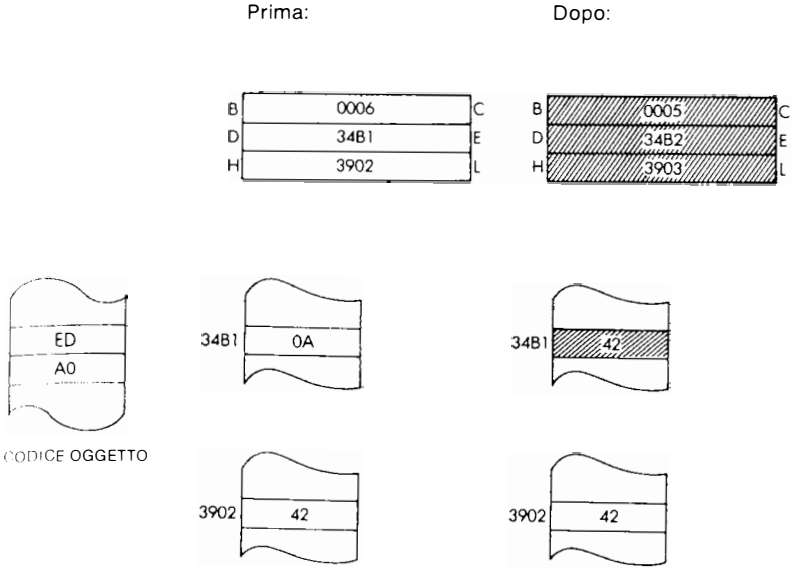
Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C
			○		X	○	

Posto a 0 se BC = 0 dopo l'esecuzione, altrimenti posto ad 1.

Esempio: LDI


Prima:
Dopo:
B
0006
C
D
34B1
E
H
3902
L
B
0005
C
D
34B2
E
H
3903
L
ED
A0
CODICE OGGETTO
34B1
0A
34B1
42
3902
42
3902
42

LDIR

Ripeti il caricamento del blocco ed incrementa.

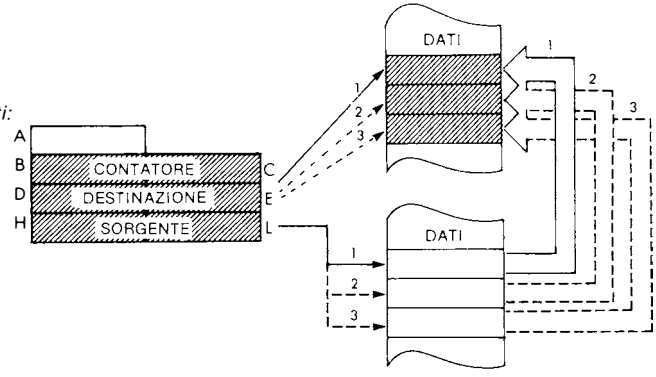
Funzione: (DE) ← (HL); DE ← DE + 1; HL ← HL + 1; BC ← BC − 1; Ripeti fino a quando BC = 0.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	0	0	byte 2: BO

Descrizione: Il contenuto della locazione di memoria indirizzata da HL, è caricato nella locazione di memoria indirizzata da DE. Poi sono incrementati sia DE che HL. BC è decrementato. Se BC ≠ 0, allora il program counter è decrementato di 2 ed è rieseguita l'istruzione.

Flusso dei dati:



Temporizzazione: Per BC ≠ 0: 5 cicli M; 21 stati T; 10.5 μsec @ 2 MHz.
Per BC = 0: 4 cicli M; 16 stati T; 8 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C
			○		○	○	

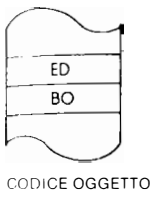
Esempio: LDIR

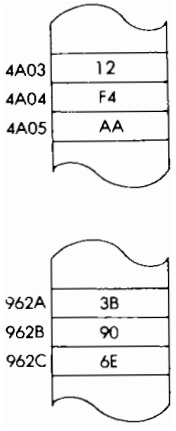
Prima:

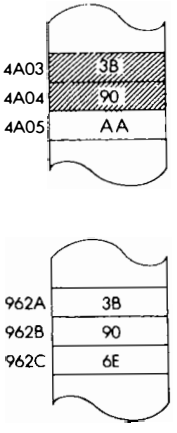
B	0002	C
D	4A03	E
H	962A	L

Dopo:

B	0000	C
D	4A05	E
H	962C	L







LD r, (HL) Carica il registro indiretto r dalla locazione di memoria (HL).

Funzione: r ← (HL)

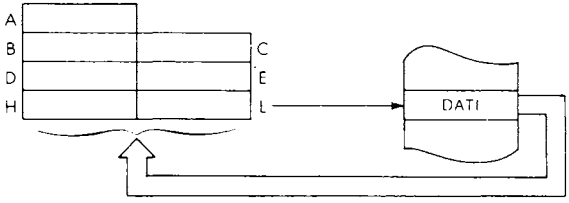
Formato:

0	1	← r →	1	1	0

Descrizione: Il contenuto della locazione di memoria indirizzata da HL, è caricato nel registro specificato.
r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 2 cicli M; 7 stati T; 3.5 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Codici del byte:

r:	A	B	C	D	E	H	L
	7E	46	4E	56	5E	66	6E

Flag:

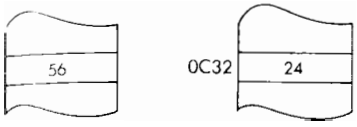
S	Z		H		P/V	N	C

(nessun effetto)

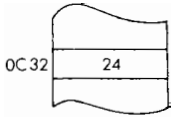
Esempio: LD D, (HL)

Prima:

Dopo:



CODICE OGGETTO



NEG

Cambia segno all'accumulatore.

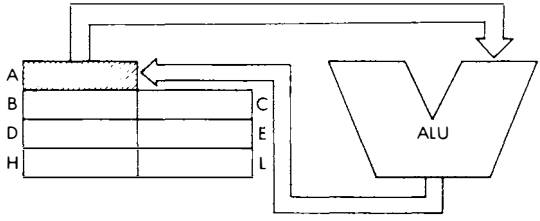
Funzione: A ← 0 − A

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	0	0	byte 2: 44

Descrizione: Il contenuto dell'accumulatore è sottratto da zero (complemento a due) e il risultato è immagazzinato di nuovo nell'accumulatore.

Flusso dei dati:



Temporizzazione: 2 cicli M; 8 stati T; 4 μsec @ 2 MHz.

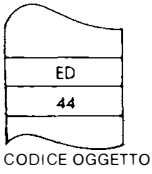
Modo d'indirizzamento: Implicito.

Flag:

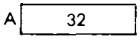
S	Z		H		P/V	N	C
●	●		●		●	1	●

C sarà posto ad 1 se A era 0 prima dell'istruzione.
P sarà posto ad 1 se A era 80H.

Esempio: NEG



Prima:



Dopo:

NOP

Nessuna operazione.

Funzione: Ritardo.

Formato:

0	0	0	0	0	0	0	0

00

Descrizione: Non viene fatto niente per 1 ciclo M.

Flusso dei dati:

A		Nessuna azione.	
B			C
D			E
H			L

Temporizzazione: 1 ciclo M; 4 stati T; 2 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

OR s Esegui l'or logico tra l'accumulatore e l'operando s.

Funzione: A ← A V s

Formato: s: può essere r, n, (HL), (IX + d), oppure (IY + d).

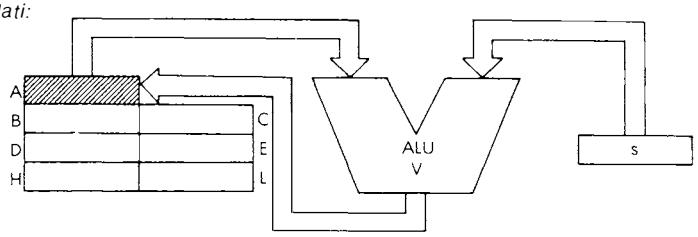
r	1	0	1	1	0	← r →			
n	1	1	1	1	0	1	1	0	byte 1: F6
	← n →								byte 2: dato immediato
(HL)	1	0	1	1	0	1	1	0	byte 1: B6
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	0	1	1	0	1	1	0	byte 2: B6
	← d →								byte 3: valore di offset
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	0	1	1	0	1	1	0	byte 2: B6
	← d →								byte 3: valore di offset

r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: Viene eseguito l'or logico dell'accumulatore e dell'operando specificato, il risultato è immagazzinato nell'accumulatore. s è definito nella descrizione delle analoghe istruzioni ADD.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @2 MHz:*
r	1	4	4
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Modo d'indirizzamento: r: implicito; n: immediato; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: OR r

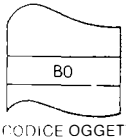
r:

A	B	C	D	E	H	L
B7	B0	B1	B2	B3	B4	B5

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	○

Esempio: OR B



CODICE OGGETTO

Prima:

A	06
B	B9

Dopo:

A	BF
B	B9

OTDR — Uscita di blocco e decremento

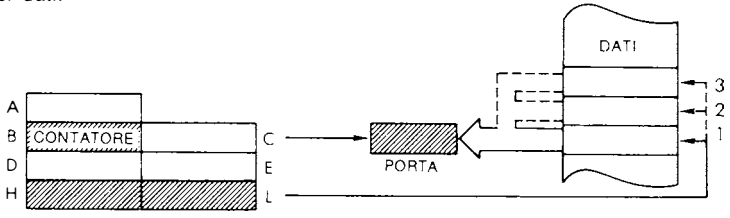
Funzione: (C) ← (HL); B B − 1; HL ← HL − 1; Ripete fino a quando B = 0.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	1	byte 2: BB

Descrizione: Il contenuto della locazione di memoria indirizzata dalla coppia di registri HL è fatta uscire al dispositivo periferico indirizzato dal contenuto del registro C. Sia il registro B che la coppia di registri HL sono poi decrementati. Se B ≠ 0, il program counter è decrementato di 2 ed è rieseguita l'istruzione. C fornisce i bit da A0 ad A7 del bus degli indirizzi. B fornisce (dopo il decremento) i bit da A8 ad A15.

Flusso dei dati:



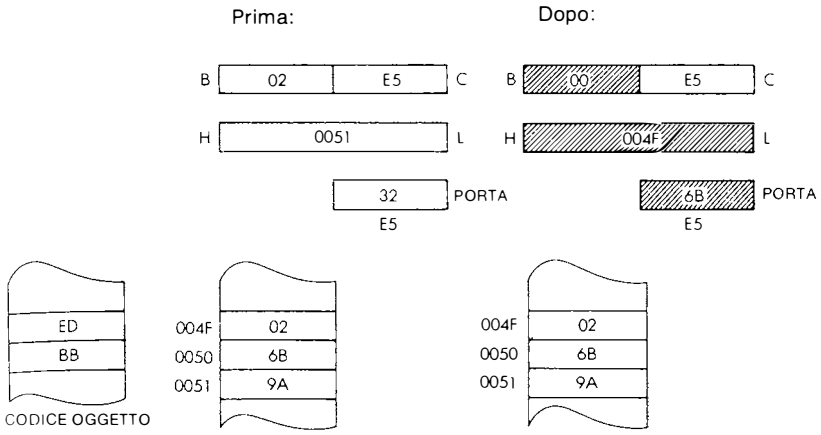
Temporizzazione: B = 0: 4 cicli M; 16 stati T; 8 µsec @ 2 MHz.
B ≠ 0: 5 cicli M; 21 stati T; 10.5 µsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	1		?		?	1	

Esempio: OTDR



OTIR Uscita di blocco ed incremento.

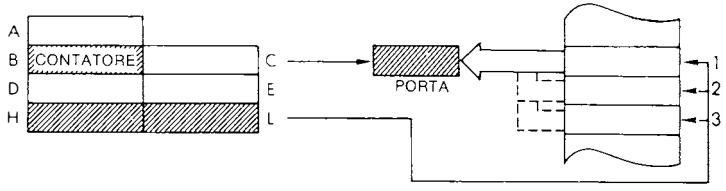
Funzione: (C) ← (HL); B ← B − 1; HL ← HL + 1; Ripete fino a quando B = 0.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	1	1	byte 2: B3

Descrizione: Il contenuto della locazione di memoria indirizzata dalla coppia di registri HL sono prelevati e mandati al dispositivo periferico indirizzato dal contenuto del registro C. Il registro B è decrementato ed è incrementata la coppia di registri HL. Se B ≠ 0, il program counter è decrementato di 2 ed è rieseguita l'istruzione. C fornisce i bit da A0 ad A7 del bus degli indirizzi. B fornisce (dopo il decremento) i bit da A8 a A15.

Flusso dei dati:



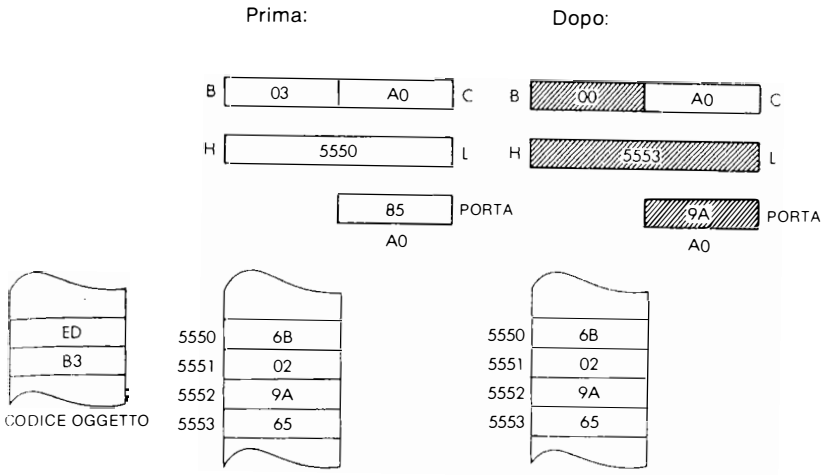
Temporizzazione: B = 0: 4 cicli M; 16 stati T; 8 μsec @ 2 MHz.
B ≠ 0: 5 cicli M; 21 stati T; 10.5 μsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	1		?		?	1	

Esempio: OTIR



OUT (C), R

Uscita del registro r sulla porta C.

Funzione: (C) ← r

Formato:

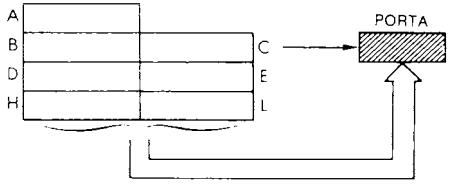
1	1	1	0	1	1	0	1	byte 1: ED
0	1	← r →			0	0	1	byte 2

Descrizione: Viene fatto uscire il contenuto del registro specificato sul dispositivo periferico indirizzato dal contenuto del registro C.
r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Il registro C fornisce i bit da A0 ad A7 del bus degli indirizzi. Il registro B fornisce i bit da A8 ad A15.

Flusso dei dati:



Temporizzazione: 3 cicli M; 12 stati T; 6 μsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Flag:

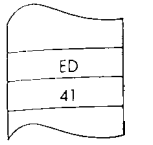
S	Z		H		P/V	N	C

(nessun effetto)

Codici del byte:

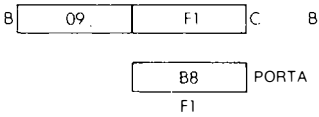
r:	A	B	C	D	E	H	L
	79	41	49	51	59	61	69

Esempio: OUT (C), B

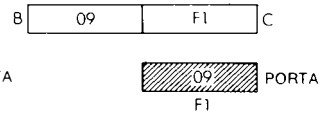


CODICE OGGETTO

Prima:



Dopo:

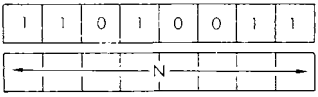


OUT (N), A

Uscita dell'accumulatore sulla porta della periferica N.

Funzione: (N) ← A

Formato:

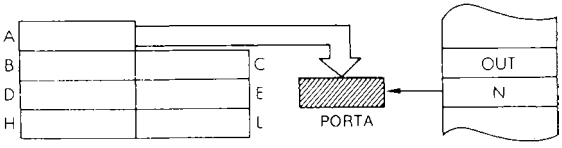


byte 1: D3

byte 2: indirizzo di porta

Descrizione: Viene fatto uscire il contenuto dell'accumulatore sul dispositivo periferico indirizzato dal contenuto della locazione di memoria che segue immediatamente il codice operativo.

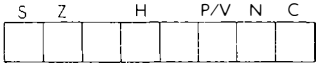
Flusso dei dati:



Temporizzazione: 3 cicli M; 11 stati T; 5.5 μsec @ 2 MHz.

Modo d'indirizzamento: Esterno.

Flag:

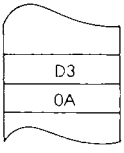


(nessun effetto)

Esempio: OUT (0A), A

Prima:

Dopo:



CODICE OGGETTO

OUTD

Uscita e decremento.

Funzione: (C) ← (HL); BC ← B − 1; HL ← HL − 1

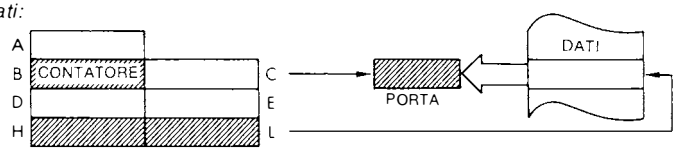
Formato:

1	1	1	0	1	1	0	1	byte 1: ED

1	0	1	0	1	0	1	1	byte 2: AB

Descrizione: Viene fatto uscire il contenuto della locazione di memoria indirizzata dalla coppia di registri HL sul dispositivo periferico indirizzato dal contenuto del registro C. Poi sono decrementati sia il registro B che la coppia di registri HL. C fornisce i bit da A0 ad A7 del bus degli indirizzi. B fornisce (dopo il decremento) da A8 ad A15.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T; 8 µsec @ 2 MHz.

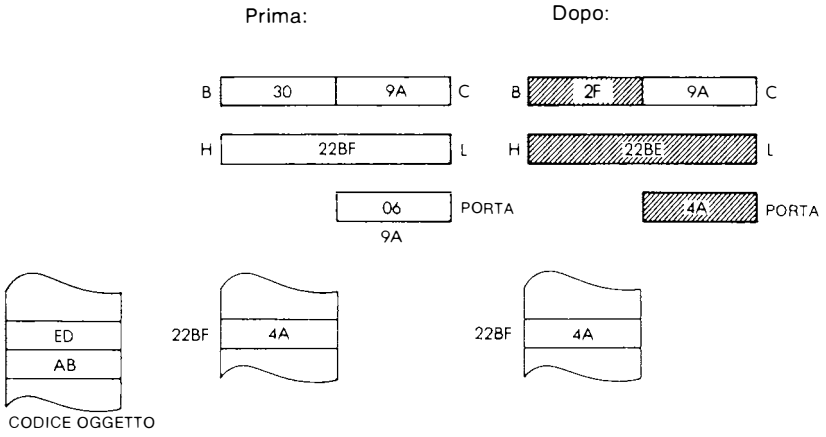
Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	X		?		?	1	

Posto ad 1 se B = 0 dopo l'esecuzione, altrimenti posto a zero.

Esempio: OUTD



OUTI

Uscita ed incremento.

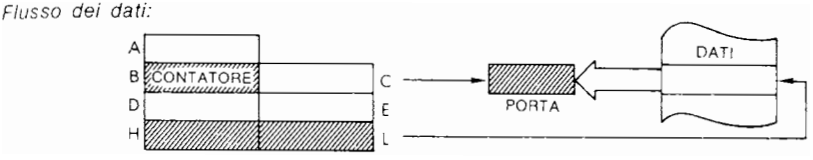
Funzione: (C) ← (HL); B ← B − 1; HL ← HL + 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	1	1	byte 2: A3

Descrizione: Viene fatto uscire il contenuto della locazione di memoria indirizzata dalla coppia di registri HL sul dispositivo periferico indirizzato dal registro C. Il registro B è decrementato ed è incrementata la coppia di registri HL.
C fornisce i bit da A0 ad A7 del bus degli indirizzi.
B (dopo il decremento) fornisce i bit da A8 ad A15.

Flusso dei dati:



Temporizzazione: 4 cicli M; 16 stati T; 8 μsec @ 2 MHz.

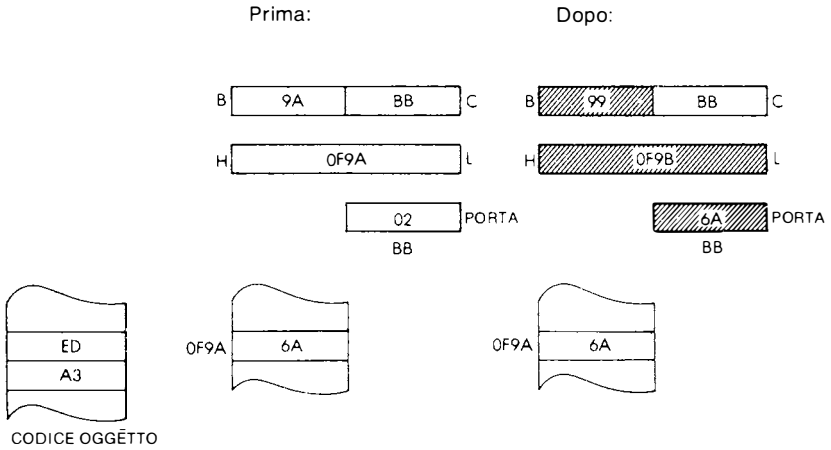
Modo d'indirizzamento: Esterno.

Flag:

S	Z		H		P/V	N	C
?	X		?		?	1	

Posto ad 1 se B = 0 dopo l'esecuzione, altrimenti posto a zero.

Esempio: OUTI



POP qq

Preleva la coppia di registri qq dallo stack.

Funzione: $qq_{basso} \leftarrow (SP)$; $qq_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

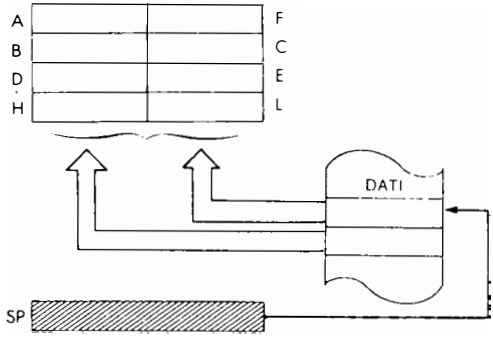
Formato:

1	1	q	q	0	0	0	1

Descrizione: Il contenuto della locazione di memoria indirizzata dal puntatore dello stack, è caricato nel byte di ordine inferiore della coppia di registri specificata e poi il puntatore dello stack è incrementato. Il contenuto della locazione di memoria ora indirizzata dal puntatore dello stack, è caricato nel byte di ordine superiore della coppia di registri, ed il puntatore dello stack è incrementato di nuovo. qq può essere ognuno di questi:

BC — 00 HL — 10
DE — 01 AF — 11

Flusso dei dati:



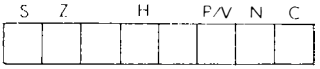
Temporizzazione: 3 cicli M; 10 stati T; 5 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Codici del byte:

qq:	BC	DE	HL	AF
	C1	D1	E1	F1

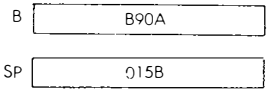
Flag:



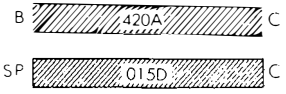
(nessun effetto)

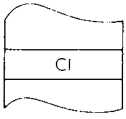
Esempio: POP BC

Prima:

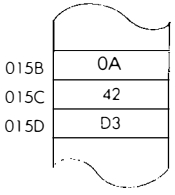


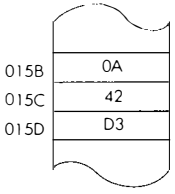
Dopo:





CODICE OGGETTO





POP IX

Preleva il registro IX dallo stack.

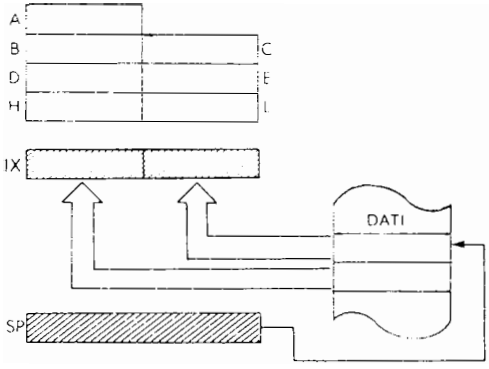
Funzione: $IX_{basso} \leftarrow (SP)$; $IX_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	0	0	1	byte 2: E1

Descrizione: Il contenuto della locazione di memoria indirizzata dal puntatore dello stack, è caricato nel byte di ordine inferiore del registro IX, ed è incrementato il puntatore dello stack. Il contenuto della locazione di memoria ora indirizzata dal puntatore dello stack, è caricato nel byte di ordine superiore del registro IX, ed il puntatore dello stack è incrementato di nuovo.

Flusso dei dati:



Temporizzazione: 4 cicli M; 14 stati T; 7 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

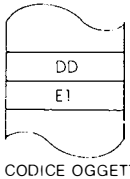
Esempio: POP IX

Prima:

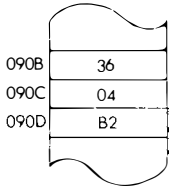
IX	0001
SP	090B

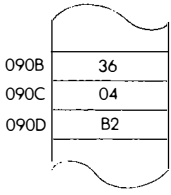
Dopo:

IX	0436
SP	090D



CODICE OGGETTO





POP IY

Preleva il registro IY dallo stack.

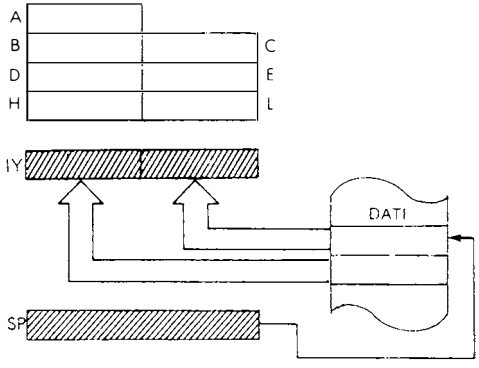
Funzione: $IY_{basso} \leftarrow (SP)$; $IY_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	0	0	1	byte 2: E1

Descrizione: Il contenuto della locazione di memoria indirizzata dal puntatore dello stack, è caricato nel byte di ordine inferiore del registro IY, e poi è incrementato il puntatore dello stack. Il contenuto della locazione di memoria ora indirizzata dal puntatore dello stack, è caricato nel byte di ordine superiore del registro IY, ed il puntatore di stack è aumentato di nuovo.

Flusso dei dati:



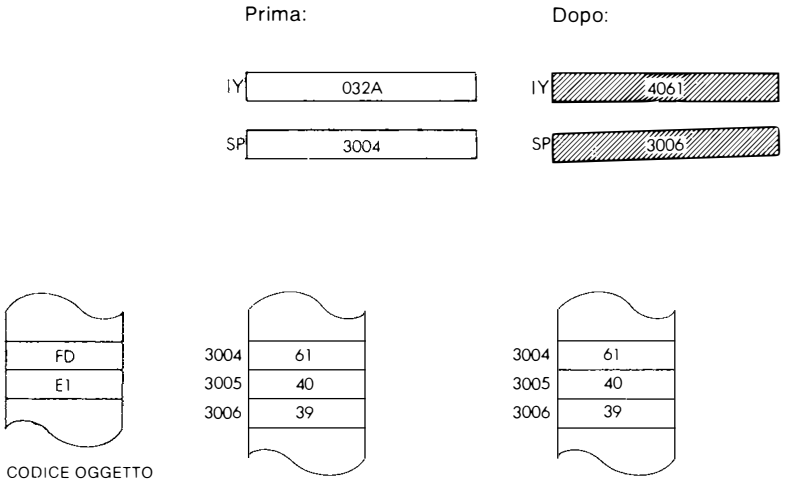
Temporizzazione: 4 cicli M; 14 stati T; 2 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z	H	P/V	N	C	
						(nessun effetto)

Esempio: POP IY



PUSH qq

Spingi la coppia di registri nello stack.

Funzione: (SP – 1) ← qq_{alto}; (SP – 2) ← qq_{basso}; SP ← SP – 2

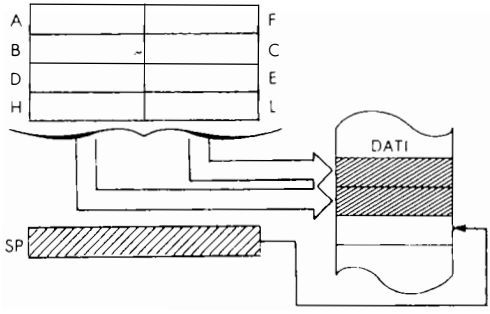
Formato:

1	1	q	q	0	1	0	1

Descrizione: Il puntatore di stack è decrementato e il contenuto del byte di ordine superiore della coppia di registri specificata, è poi caricato nella locazione di memoria indirizzata dal puntatore dello stack. Il puntatore dello stack è decrementato di nuovo e il contenuto del byte di ordine inferiore della coppia di registri è caricato nella locazione di memoria attualmente indirizzata dal puntatore dello stack. qq può essere ognuno di questi:

BC – 00	HL – 10
DE – 01	AF – 11

Flusso dei dati:



Temporizzazione: 3 cicli M; 11 stati T; 6.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Codici del byte:

qq	BC	DE	HL	AF
	C5	D5	E5	F5

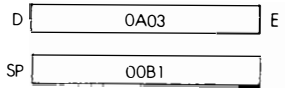
Flag:

S	Z		H		P/V	N	C

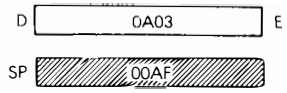
(nessun effetto)

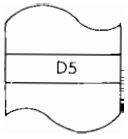
Esempio: PUSH DE

Prima:

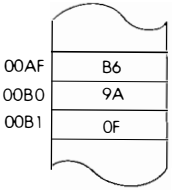


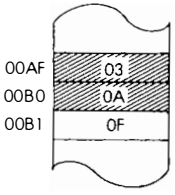
Dopo:





CODICE OGGETTO





PUSH IX

Spingi IX nello stack.

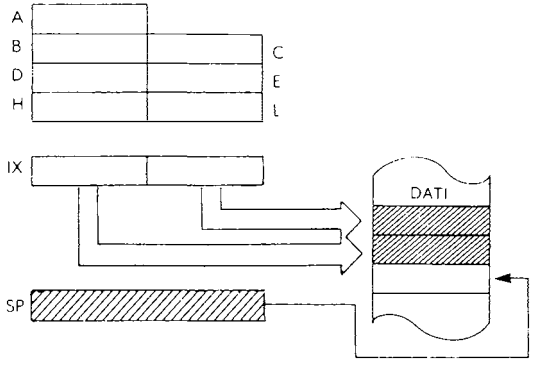
Funzione: (SP − 1) ← IX_{alto}; (SP − 2) ← IX_{basso}; SP ← SP − 2

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	1	0	1	byte 2: E5

Descrizione: Il puntatore dello stack è decrementato, e il contenuto dell'ordine superiore del registro IX è caricato nella locazione di memoria indirizzata dal puntatore dello stack. Il puntatore dello stack è decrementato di nuovo e poi il contenuto dell'ordine inferiore del registro IX è caricato nella locazione di memoria indirizzata dal puntatore dello stack.

Flusso dei dati:



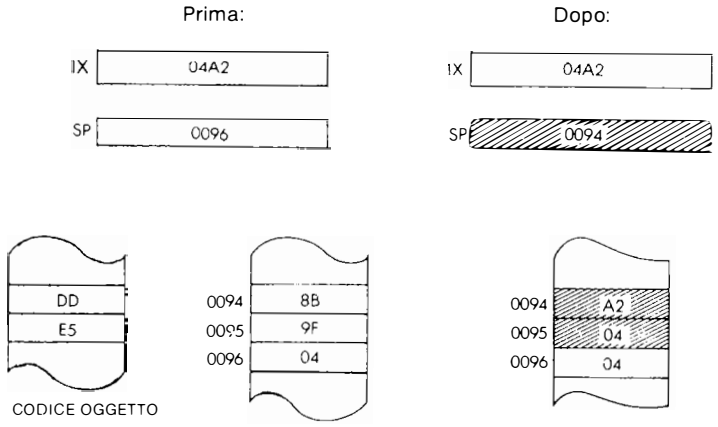
Temporizzazione: 4 cicli M; 15 stati T; 7.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

Esempio: PUSH IX



PUSH IY

Spingi IY nello stack.

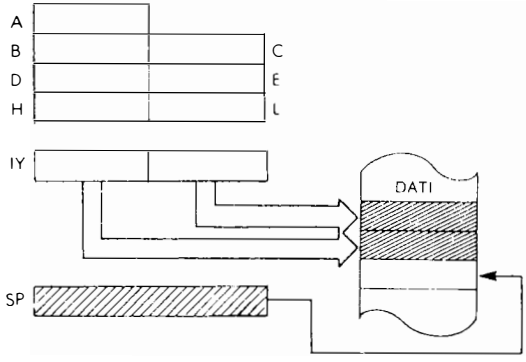
Funzione: (SP – 1) ← IYalto ; (SP – 2) ← IYbasso ; SP ← SP – 2.

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	1	0	1	byte 2: E5

Descrizione: Il puntatore dello stack è decrementato ed il contenuto del byte di ordine superiore del registro IY è caricato nella locazione di memoria indirizzata dal puntatore dello stack. Il puntatore dello stack è decrementato di nuovo ed il contenuto del byte di ordine inferiore del registro IY è caricato nella locazione di memoria indirizzata dal puntatore dello stack.

Flusso dei dati:



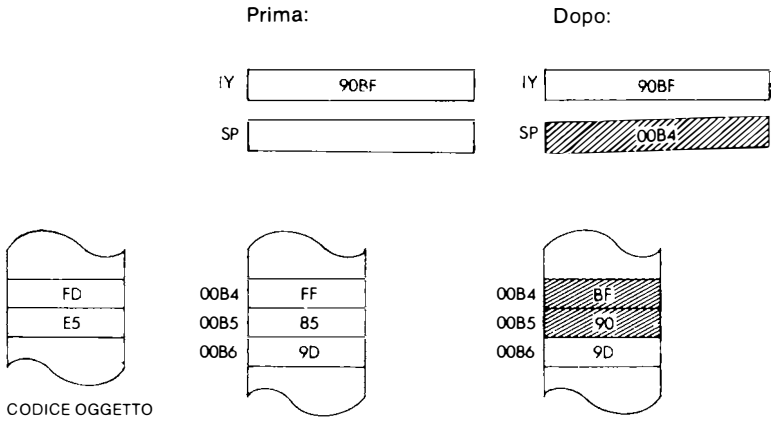
Temporizzazione: 3 cicli M; 15 stati T; 7.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

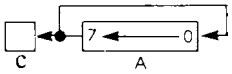
Esempio: PUSH IY



RLCA

Ruota l'accumulatore a sinistra con ramificazione nel riporto.

Funzione:



Formato:

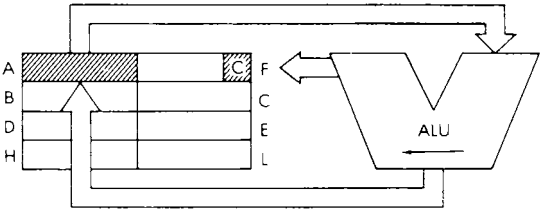
0	0	0	0	0	1	1	1

07

Descrizione:

Il contenuto dell'accumulatore è ruotato a sinistra di una posizione di bit. Il contenuto originale del bit 7 è spostato al flag di riporto così come al bit 0.

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 μsec @ 2 MHz.

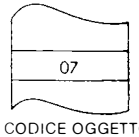
Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C
			○			○	●

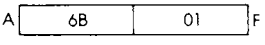
C è posizionato dal bit 7 di A.

Esempio: RLCA



CODICE OGGETTO

Prima:



Dopo:

RES b, s

Poni a 0 il bit b dell'operando s.

Funzione: $s_b - 0$

Formato:

s:									
r	1	1	0	0	1	0	1	1	byte 1: CB
	1	0	← b →			← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	1	0	← b →			1	1	0	byte 2
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	1	0	← b →			1	1	0	byte 4
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	1	0	← b →			1	1	0	byte 4

b può essere ognuno di questi:

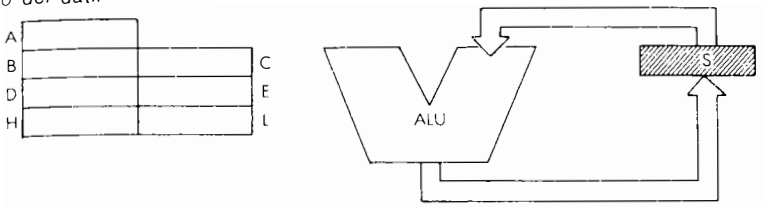
0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descrizione: Il bit specificato della locazione determinata da s è posto a 0. s è definito nella descrizione delle analoghe istruzioni BIT.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: RES b, r

	r:	A	B	C	D	E	H	L
	b:							
CB	0	87	80	81	82	83	84	85
	1	8F	88	89	8A	8B	8C	8D
	2	97	90	91	92	93	94	95
	3	9F	98	99	9A	9B	9C	9D
	4	A7	A0	A1	A2	A0	A4	A5
	5	AF	A8	A9	AA	AB	AC	AD
	6	B7	B0	B1	B2	B3	B4	B5
	7	BF	B8	B9	BA	BB	BC	BD

RES b, (HL)

	b:	0	1	2	3	4	5	6	7
CB		86	BE	96	9E	A6	AE	B6	BE

RES b, (IX + d)

RES b, (IY + d)

		b: 0	1	2	3	4	5	6	7
DD— / FD—	CB —	86	8E	96	9E	A6	AE	B6	BE

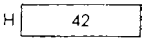
Flag:

S	Z		H		P/V	N	C

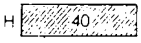
(nessun effetto)

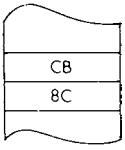
Esempi: RES 1, H

Prima:



Dopo:





CODICE OGGETTO

RET Ritorna dalla subroutine

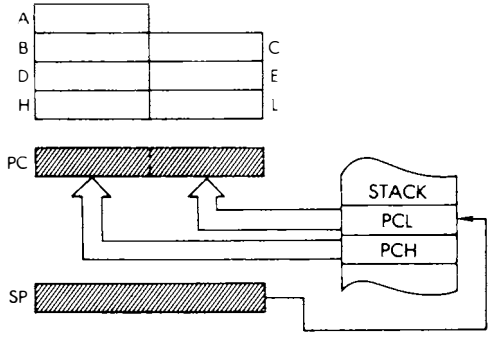
Funzione: $PC_{basso} \leftarrow (SP)$; $PC_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Formato:

1	1	0	0	1	0	0	1	C9

Descrizione: Il PC è prelevato dallo stack come descritto per le istruzioni POP. L'istruzione successiva eseguita, è quella alla locazione puntata dal PC.

Flusso dei dati



Temporizzazione: 3 cicli M; 10 stati T; 5 µsec @ 2 MHz.

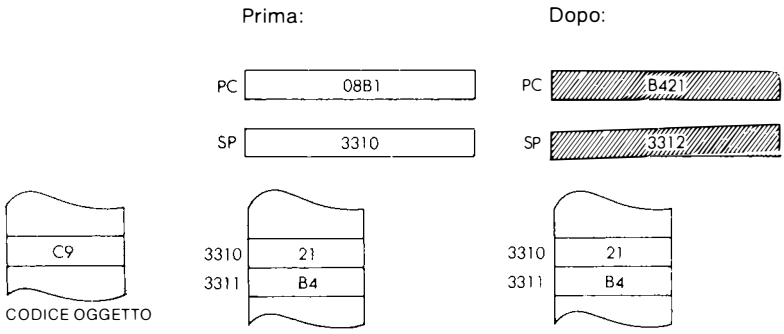
Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: RET


Prima:
Dopo:
PC
08B1
SP
3310
PC
B421
SP
3312
C9
CODICE OGGETTO
3310
21
3311
B4
3310
21
3311
B4

RET cc

Ritorna in modo condizionato dalla subroutine.

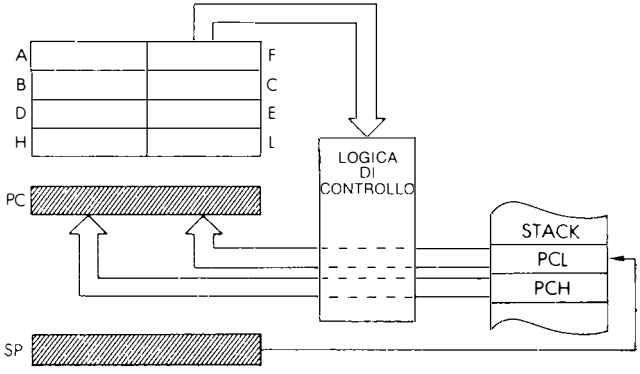
Funzione: Se cc è vero: $PC_{basso} \leftarrow (SP)$; $PC_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Formato:

1	1	← cc →	0	0	0

Descrizione: Se è soddisfatta la condizione, il contenuto del program counter è prelevato dallo stack come descritto per le istruzioni POP.
L'istruzione successiva è prelevata dall'indirizzo nel PC.
Se non è soddisfatta la condizione, l'esecuzione dell'istruzione continua in sequenza.

Flusso dei dati:



cc può essere ognuno di questi:

NZ – 000	PO – 100
Z – 001	PE – 101
NC – 010	P – 110
C – 011	M – 111

Temporizzazione: Condizione soddisfatta: 3 cicli M; 11 stati T; 6.5 μsec @ 2 MHz.
Condizione non soddisfatta: 1 ciclo M; 5 stati T: 2,5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Codici del byte:

CC:	NZ	Z	NC	C	PO	PE	P	M
	C0	C8	D0	D8	E0	E8	F0	F8

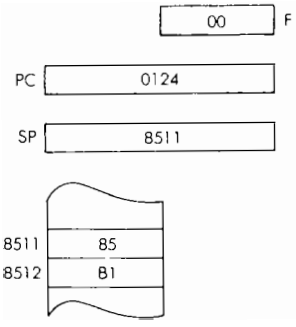
Flag:

S	Z		H		P/V	N	C

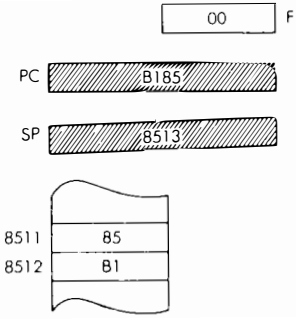
(nessun effetto)

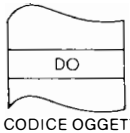
Esempio: RET NC

Prima:



Dopo:





CODICE OGGETTO

RETI

Ritorna dall'interrupt.

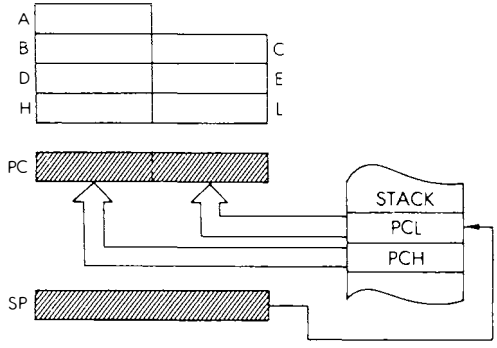
Funzione: $PC_{basso} \leftarrow (SP)$; $PC_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	0	1	byte 2: 4D

Descrizione: Il program counter è prelevato dallo stack come descritto per le istruzioni POP. Questa istruzione è riconosciuta dai dispositivi periferici Zilog come la parte finale di una routine di servizio periferica così da permettere un corretto controllo di interrupt regolati da priorità a nido. Una istruzione EI deve essere eseguita prima di RETI per re-abilitare gli interrupt.

Flusso dei dati:



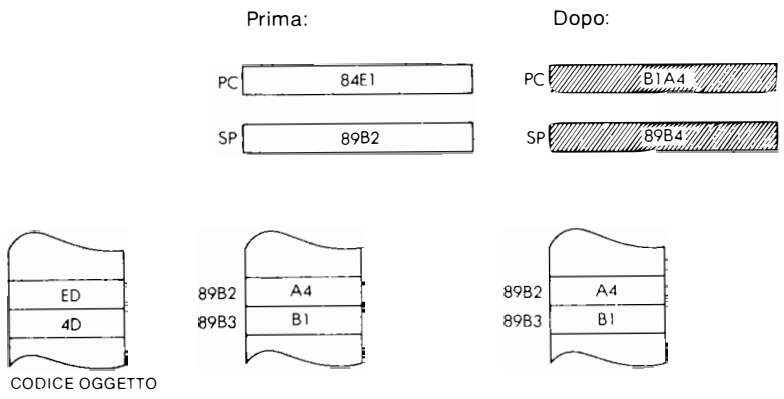
Temporizzazione: 4 cicli M; 14 stati T; 7 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C	
								(nessun effetto)

Esempio: RETI



RETN

Ritorna dall'interrupt non mascherabile.

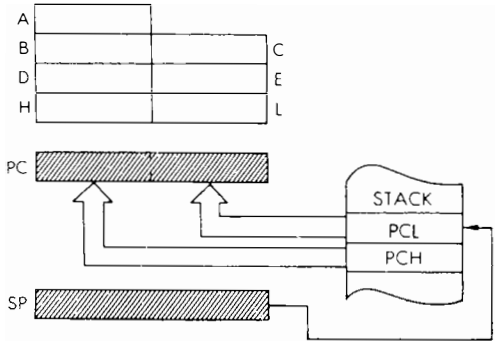
Funzione: $PC_{basso} \leftarrow (SP)$; $PC_{alto} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$; $IFF1 \leftarrow IFF2$

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	0	1	byte 2: 45

Descrizione: Il program counter è prelevato dallo stack come descritto per le istruzioni POP. Poi il contenuto dell'IFF2 (flip-flop d'immagazzinamento) è riscritto nell'IFF1 per ri-memorizzare lo stato del flag d'interrupt prima dell'interrupt non mascherabile.

Flusso dei dati:



Temporizzazione: 4 cicli M; 14 stati T; 7 µsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

Esempio: RETN

Prima:

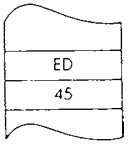
PC A5F1

SP 8B4C

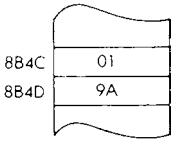
Dopo:

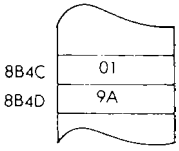
PC 9A01

SP 8B4E



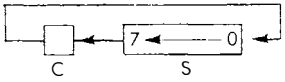
CODICE OGGETTO





RL s Ruota a sinistra l'operando s attraverso il riporto.

Funzione:



Formato:

	s:								
	r	1	1	0	0	1	0	1	1	byte 1: CB
		0	0	0	1	0	← r →			byte 2
	(HL)	1	1	0	0	1	0	1	1	byte 1: CB
		0	0	0	1	0	1	1	0	byte 2: 16
	(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
		1	1	0	0	1	0	1	1	byte 2: CB
		← d →								byte 3: valore di offset
		0	0	0	1	0	1	1	0	byte 4: 16
	(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
		1	1	0	0	1	0	1	1	byte 2: CB
		← d →								byte 3: valore di offset
		0	0	0	1	0	1	1	0	byte 4: 16

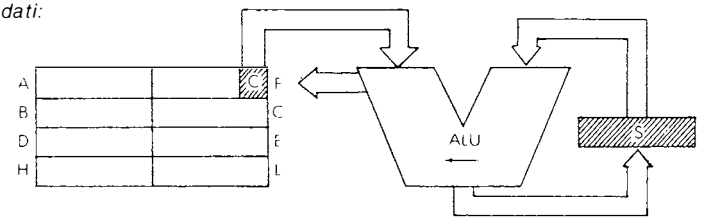
r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descrizione:

Il contenuto della locazione dell'operando specifico è spostato a sinistra di un posto di bit.
Il contenuto del flag di riporto è spostato al bit 0 e il contenuto del bit 7 è spostato al flag di riporto. Il risultato finale è immagazzinato di nuovo nella locazione originale. s è definito nella descrizione delle analoghe istruzioni RLC.

Flusso dei dati:

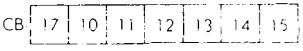


Temporizzazione:

s:	*cicli M:*	*stati T:*	µsec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

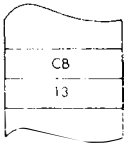
Codici del byte: RL r:



Flag:

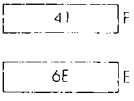
C è posto ad uno dal bit 7 di sorgente.

Esempio: RL E

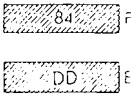


CODICE OGGETTO

Prima:



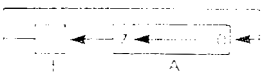
Dopo:



RLA

Ruota l'accumulatore a sinistra attraverso il flag di riporto.

Funzione:



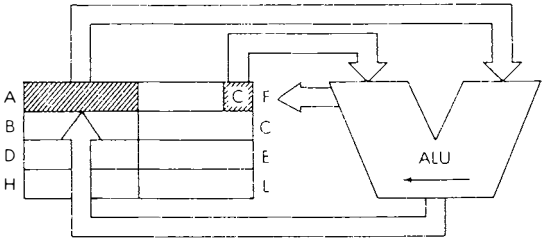
Formato:

0	0	0	1	0	1	1	1

17

Descrizione: Il contenuto dell'accumulatore è spostato a sinistra di una posizione di bit. Il contenuto del flag di riporto è mosso nel bit 0 e il contenuto originale del bit 7 è mosso nel flag di riporto (rotazione a 9 bit).

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

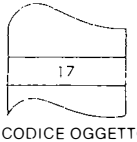
S	Z	H	P/V	N	C
		○		○	●

C è posto ad uno dal bit 7 di A.

Esempio: RLA

Prima:

Dopo:

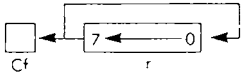


CODICE OGGETTO

RLC r

Ruota a sinistra la locazione di memoria (HL) con ramificazione nel riporto.

Funzione:



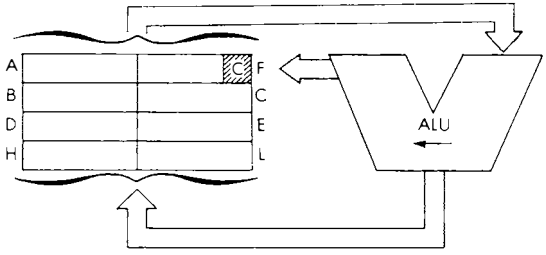
Formato:

1	1	0	0	1	0	1	1	byte 1: CB
0	0	0	0	0	← r →			byte 2

Descrizione: Il contenuto del registro specificato è ruotato a sinistra. Il contenuto originale del bit 7 è mosso al flag di riporto così come il bit 0. r può essere ognuno di questi:

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Flusso dei dati:



Temporizzazione: 2 cicli M; 8 stati T; 4 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Codici del byte: r:

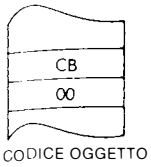
	A	B	C	D	E	H	L
CB-	07	00	01	02	03	04	05

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

C è posto ad uno dal bit 7 del registro di sorgente.

Esempio: RLC B



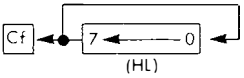
Prima:

Dopo:

RLC (HL)

Ruota a sinistra la locazione di memoria (HL) con ramificazione nel riporto.

Funzione:



Formato:

1	1	0	0	1	0	1	1

byte 1: CB

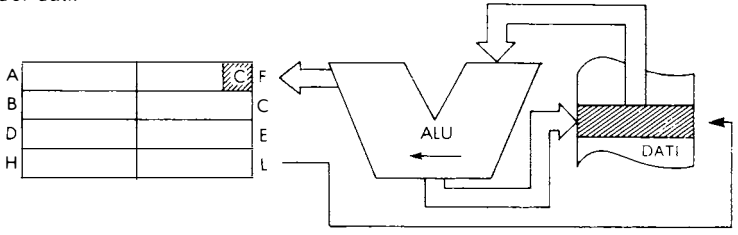
0	0	0	0	0	1	1	0

byte 2: 06

Descrizione:

Il contenuto della locazione di memoria indirizzata dal contenuto della coppia di registri HL, è ruotata a sinistra di una posizione bit e il risultato è immagazzinato nuovamente in quella locazione. Il contenuto del bit 7 è trasferito nel flag di riporto e nel bit 0.

Flusso dei dati:



Temporizzazione: 4 cicli M; 15 stati T; 7.5 µsec @ 2 MHz.

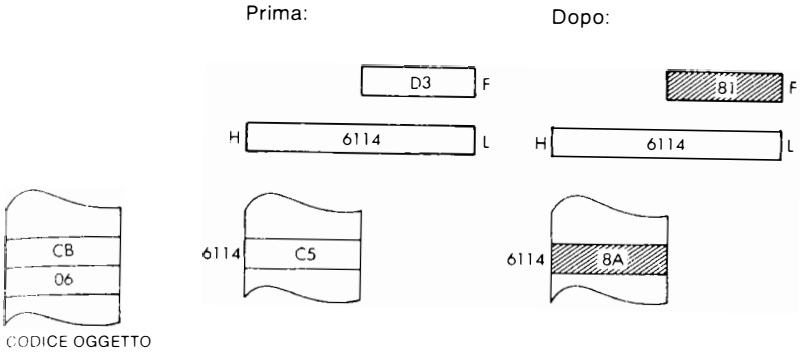
Modo d'indirizzamento: Indiretto.

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

C è posto ad uno dal bit 7 della locazione di memoria.

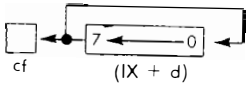
Esempio: RLC (HL)



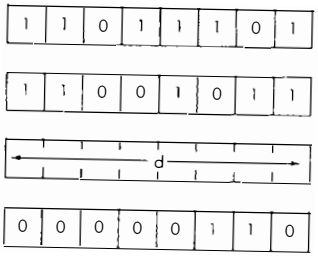
RLC (IX + d)

Ruota a sinistra la locazione di memoria (IX + d) con ramificazione nel riporto.

Funzione:



Formato:



byte 1: DD

byte 2: CB

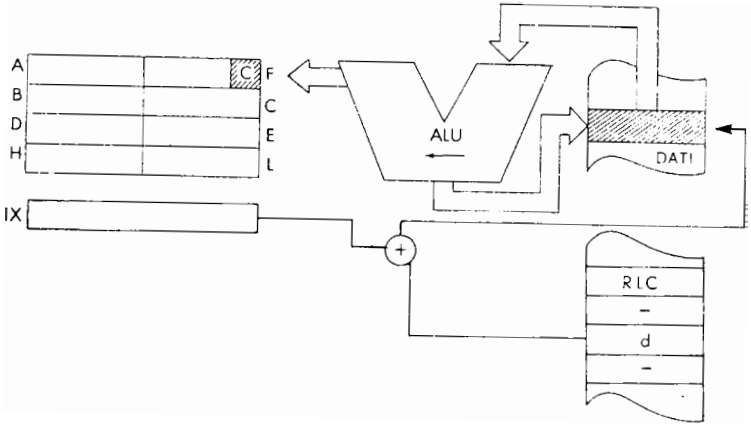
byte 3: valore di offset

byte 4: 06

Descrizione:

Il contenuto della locazione di memoria indirizzata dal contenuto del registro IX più il dato valore di offset, è ruotato a sinistra e il risultato è immagazzinato di nuovo in quella locazione. Il contenuto del bit 7 è trasferito nel flag di riporto e nel bit 0.

Flusso dei dati:



Temporizzazione: 6 cicli M; 23 stati T; 11.5 μsec @ 2 MHz.

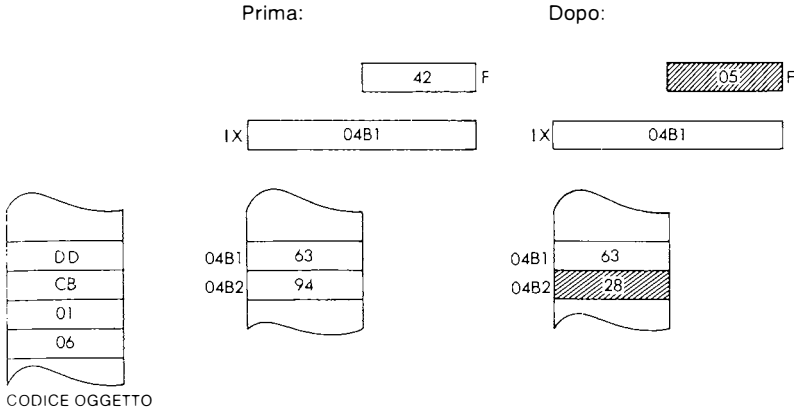
Modo d'indirizzamento: Indicizzato.

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

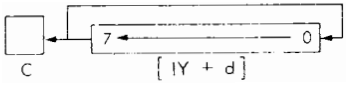
C è posto ad uno dal bit 7 della locazione di memoria.

Esempio: RLC (IX + 1)

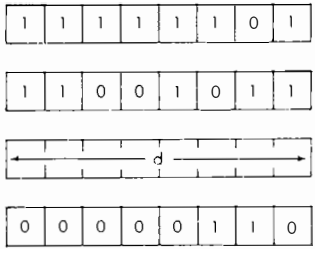


RLC (IY + d) Ruota a sinistra la locazione di memoria (IY + d) con riporto.

Funzione:



Formato:



byte 1: FD

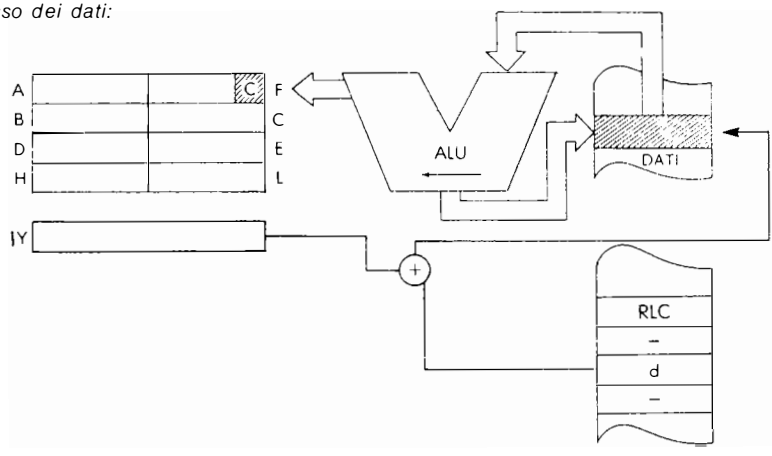
byte 2: CB

byte 3: valore di offset

byte 4: 06

Descrizione: Il contenuto della locazione di memoria indirizzata dal contenuto del registro IY più il dato valore di offset, è ruotato a sinistra e il risultato è immagazzinato di nuovo in quella locazione. Il contenuto del bit 7 è trasferito nel flag di riporto e nel bit 0.

Flusso dei dati:



Temporizzazione: 6 cicli M; 23 stati T; 11.5 μsec @ 2 MHz.

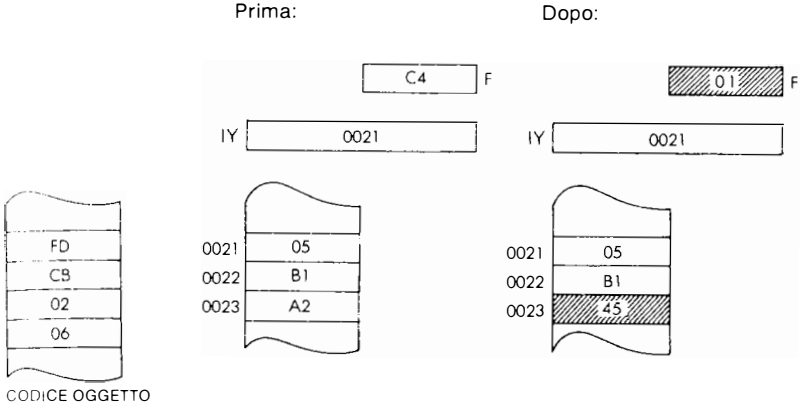
Modo d'indirizzamento: Indicizzato.

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

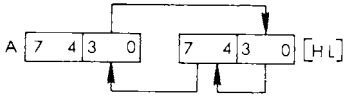
C è posto ad uno dal bit 7 della locazione di memoria.

Esempio: RLC (IY + 2)



RLD Rotazione decimale a sinistra.

Funzione:

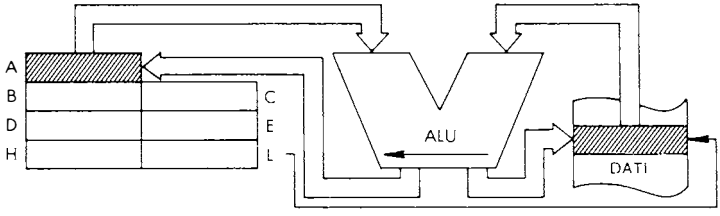


Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	1	0	1	1	1	1	byte 2: 6F

Descrizione: I 4 bit di ordine inferiore della locazione di memoria indirizzata dal contenuto di HL sono spostati alle posizioni di bit di ordine superiore di quella stessa locazione. I 4 bit d'ordine superiore sono spostati ai 4 bit d'ordine inferiore dell'accumulatore. L'ordine inferiore dell'accumulatore è spostato ai 4 bit d'ordine inferiore della locazione di memoria originariamente specificata. Tutte queste operazioni avvengono simultaneamente.

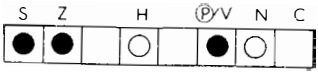
Flusso dei dati:



Temporizzazione: 5 cicli M; 18 stati T; 9 µsec @ 2 MHz.

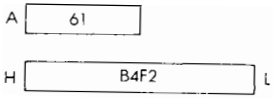
Modo d'indirizzamento: Indiretto.

Flag:

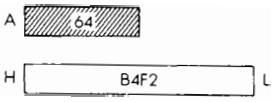


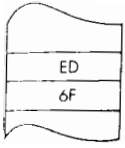
Esempio: RLD

Prima:

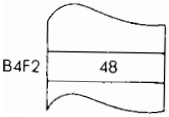


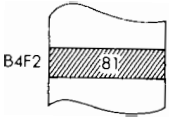
Dopo:





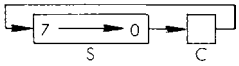
CODICE OGGETTO





RR s Ruota s a destra attraverso il riporto.

Funzione:



Formato:

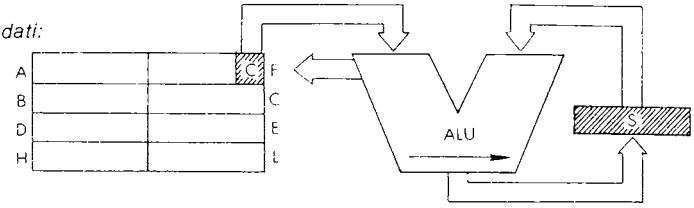
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	1	r			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	1	1	1	0	byte 2: 1E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	d								byte 3: valore di offset
	0	0	0	1	1	1	1	0	byte 4: 1E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	d								byte 3: valore di offset
	0	0	0	1	1	1	1	0	byte 4: 1E

r può essere ognuno di questi:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descrizione: Il contenuto della locazione determinata dall'operando specificato, è spostato a destra. Il contenuto del flag di riporto è spostato al bit 7 e il contenuto del bit 0 è spostato al flag di riporto. Il risultato finale è immagazzinato di nuovo nella locazione originale. s è definito nella descrizione delle analoghe istruzioni RLC.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: RR r:

r:	A	B	C	D	E	H	L
CB-	1F	18	19	1A	1B	1C	1D

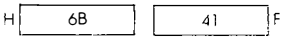
Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

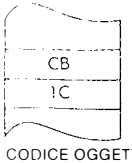
C è posto ad uno dal bit 0 dei dati di sorgente.

Esempio: RR H

Prima:



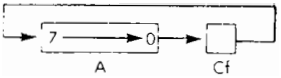
Dopo:



CODICE OGGETTO

RRA Ruota l'accumulatore a destra attraverso il riporto.

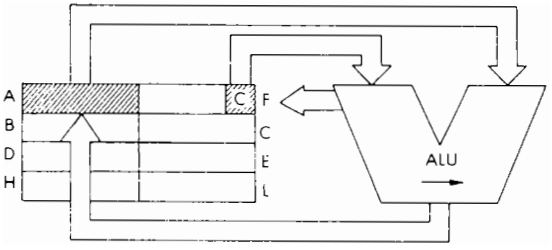
Funzione:



Formato:

Descrizione: Il contenuto dell'accumulatore è spostato a destra di una posizione di bit. Il contenuto del flag di riporto è spostato al bit 7 e il contenuto del bit 0 è spostato al flag di riporto (rotazione a 9 bit).

Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

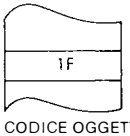
S	Z		H		P/V	N	C
			○			○	●

C è posto ad uno dal bit 0 di A.

Esempio: RRA

Prima:

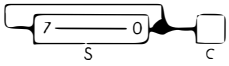
Dopo:



CODICE OGGETTO

RRC s Ruota a destra s con ramificazione nel riporto.

Funzione:



Formato: s: s è ognuno di r, (HL), (IX + d), (IY + d).

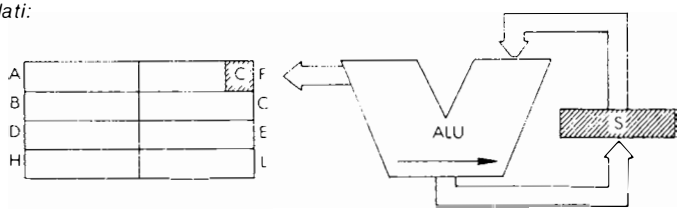
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	0	1	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	0	1	1	1	0	byte 2: 0E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	0	0	1	1	1	0	byte 4: 0E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	0	0	1	1	1	0	byte 4: 0E

r può essere ognuno di questi:

A — 111
B — 000
C — 001
D — 010
E — 011
H — 100
L — 101

Descrizione: Il contenuto della locazione determinata dall'operando specificato, è ruotato a destra ed il risultato è immagazzinato di nuovo nella locazione originale. Il contenuto del bit 0 è spostato al flag di riporto cosi come al bit 7. s è definito nella descrizione delle analoghe istruzioni RLC.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec: @2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: RRC r

r:	A	B	C	D	E	H	L
CB	0F	08	09	0A	0B	0C	0D

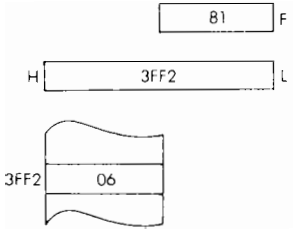
Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

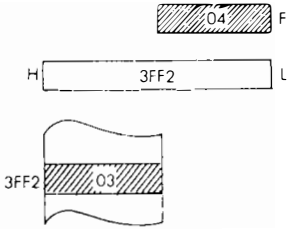
C è posto ad uno dal bit 0 dei dati di sorgente.

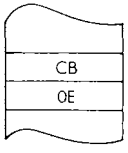
Esempio: RRC (HL)

Prima:



Dopo:



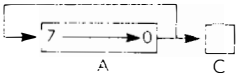


CODICE OGGETTO

RRCA

Ruota l'accumulatore a destra con ramificazione nel riporto.

Funzione:

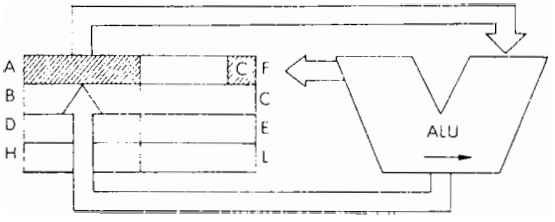


Formato:

Descrizione:

Il contenuto dell'accumulatore è ruotato a destra di una posizione di bit. Il contenuto del bit 0 è spostato al flag di riporto come pure al bit 7.

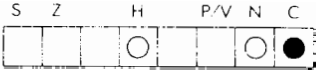
Flusso dei dati:



Temporizzazione: 1 ciclo M; 4 stati T; 2 µsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

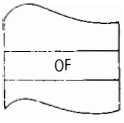
Flag:



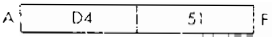
C è posto ad uno dal bit 0 di A.

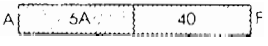
Esempio: RRCA

Prima: Dopo:



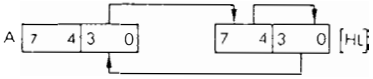
CODICE OGGETTO





RRD Ruotazione decimale a destra.

Funzione:

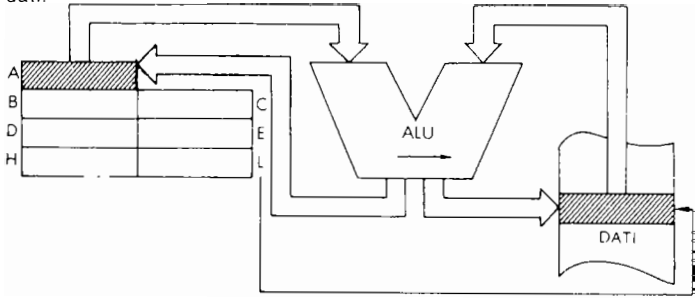


Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	1	0	0	1	1	1	byte 2: 67

Descrizione: I 4 bit d'ordine superiore della locazione di memoria indirizzata dal contenuto della coppia di registri HL, sono spostati ai 4 bit d'ordine inferiore di quella locazione. I 4 bit di ordine inferiore sono spostati ai 4 bit d'ordine inferiore dell'accumulatore. I bit d'ordine inferiore dell'accumulatore sono spostati alle 4 posizioni di bit d'ordine superiore della locazione di memoria originariamente specificata.

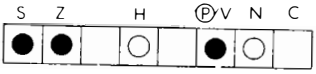
Flusso dei dati:



Temporizzazione: 5 cicli M; 18 stati T; 9 µsec @ 2 MHz.

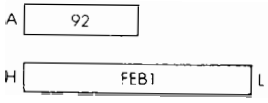
Modo d'indirizzamento: Indiretto.

Flag:

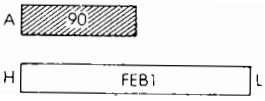


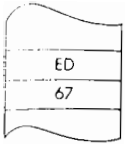
Esempio: RRD

Prima:

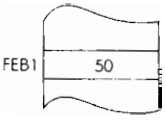


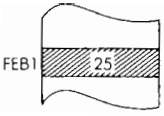
Dopo:





CODICE OGGETTO





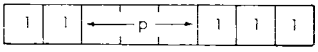
RST p

Ricomincia a p.

Funzione:

(SP − 1) ← PC_{alto}; (SP − 2) ← PC_{basso}; SP ← SP − 2; PC_{alto} ← 0; PC_{basso} ← p.

Formato:



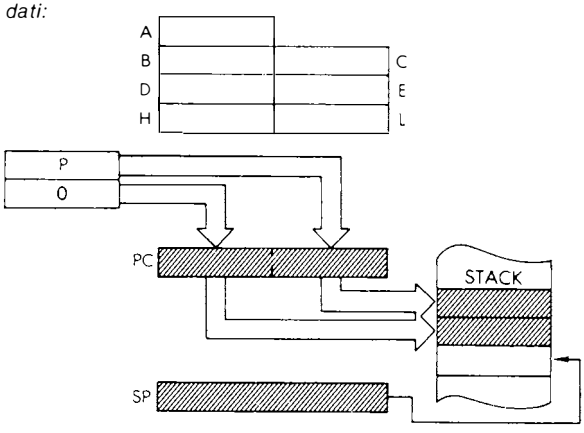
Descrizione:

Il contenuto del program counter è spinto nello stack come descritto per le istruzioni PUSH. Il valore specificato per p è poi caricato nel PC la prossima istruzione è prelevata da questo nuovo indirizzo. p può essere ognuno di questi:

00H – 000	20H – 100
08H – 001	28H – 101
10H – 010	30H – 110
18H – 011	38H – 111

Questa istruzione compie un salto ad ognuno degli otto indirizzi d'inizio della memoria inferiore e richiede solo un byte singolo. Può essere usata come una risposta veloce ed un interrupt.

Flusso dei dati:



Temporizzazione: 3 cicli M; 11 stati T; 5.5 μsec @ 2 MHz.

Modo d'indirizzamento: Indiretto.

Codici del byte:

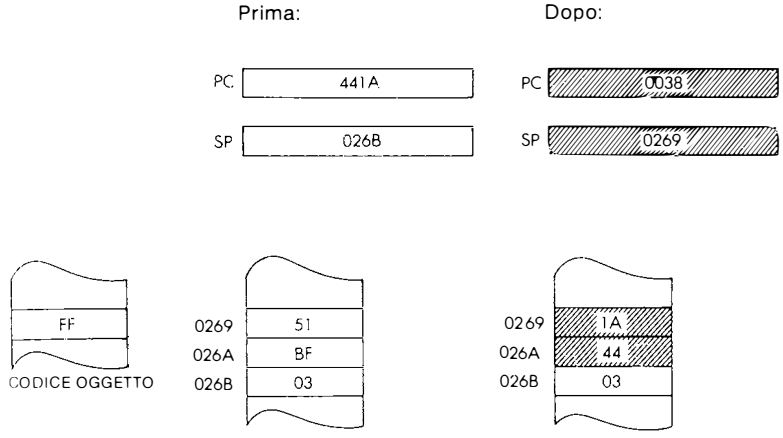
p:	00	08	10	18	20	28	30	38
	C7	CF	D7	DF	E7	EF	F7	FF

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

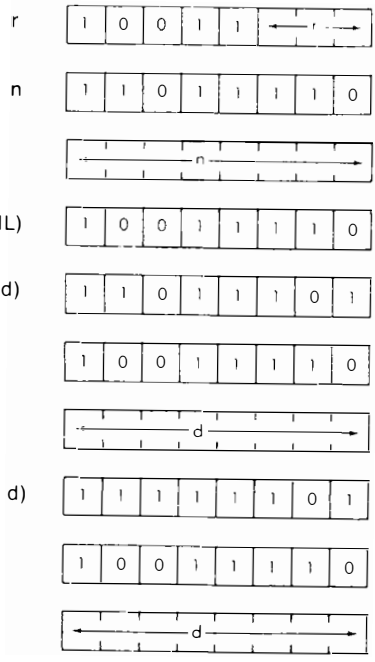
Esempio: RST 38H



SBC A, s Sottrazione con prestito tra accumulatore e l'operando specificato.

Funzione: A ← A − s − C

Formato: s: può essere r, n, (HL), (IX + d), oppure (IY + d)

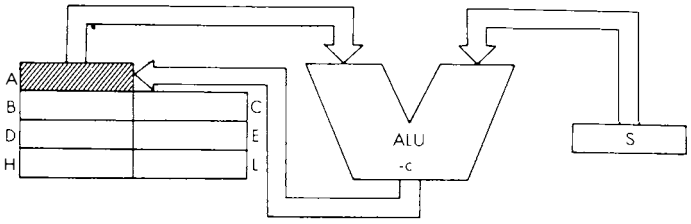


r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: L'operando specificato s, sommato con il contenuto del flag di riporto, è sottratto dal contenuto dell'accumulatore, ed il risultato è posto nell'accumulatore. s è definito nella descrizione delle analoghe istruzioni ADD.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Modo d'indirizzamento: r: implicito; n: immediato; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: SBC A, r

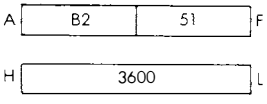
r: A	B	C	D	E	H	L
9F	98	99	9A	9B	9C	9D

Flag:

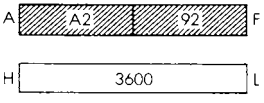
S	Z		H		P/V	N	C
●	●		●		●	1	●

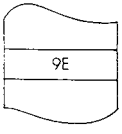
Esempio: SBC A, (HL)

Prima:

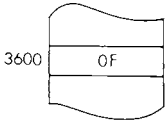


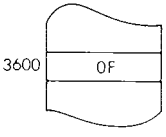
Dopo:





CODICE OGGETTO





SBC HL, ss

Sottrazione con prestito tra HL e la coppia di registro ss.

Funzione: HL ← HL − ss − C

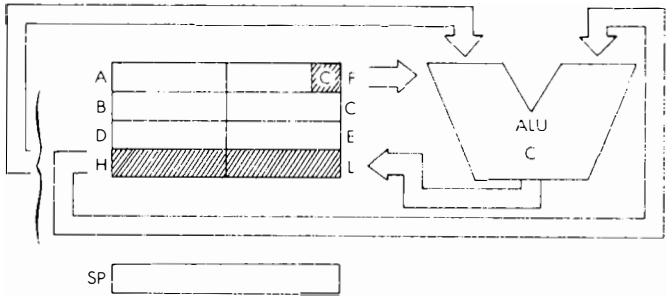
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	S	S	0	0	1	0	byte 2

Descrizione: Il contenuto della coppia di registri specificata più il contenuto del flag di riporto, è sottratto dal contenuto della coppia di registri HL ed il risultato è immagazzinato di nuovo in HL. ss può essere ognuno di questi:

BC – 00 HL – 10
DE – 01 SP – 11

Flusso dei dati:



Temporizzazione: 4 cicli M; 15 stati T; 7.5 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Codici del byte:

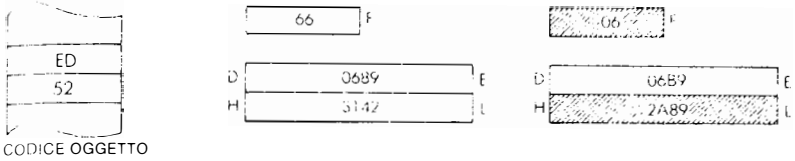
SS:	BC	DE	HL	SP
ED	42	52	62	72

Flag:

H è posto ad uno se si prende a prestito dal bit 12.
C è posto ad uno se si prende a prestito.

Esempio: SBC HL, DE

Prima: Dopo:



SCF Poni ad 1 il flag di riporto.

Funzione: C ← 1

Formato:

0	0	1	1	0	1	1	1

37

Descrizione: Il flag di riporto è posto ad 1.

Temporizzazione: 1 ciclo M; 4 stati T; 2 μsec @ 2 MHz.

Modo d'indirizzamento: Implicito.

Flag:

S	Z		H		P/V	N	C
			○			○	1

SET b, s

Poni ad 1 il bit b dell'operando.

Funzione: $s_b \leftarrow 1$

Formato: s:

r	1	1	0	0	1	0	1	1	byte 1: CB
	1	1	b			r			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	1	1	b			1	1	0	byte 2
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	d								byte 3: valore di offset
	1	1	b			1	1	0	byte 4
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	d								byte 3: valore di offset
	1	1	b			1	1	0	byte 4

r può essere ognuno di questi:

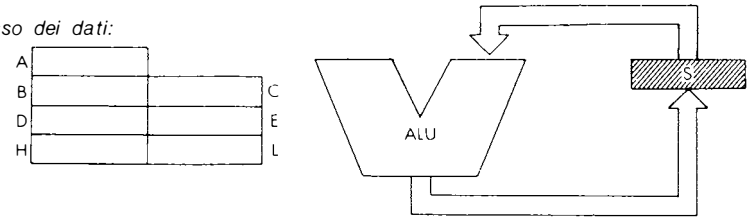
A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

b può essere ognuno di questi:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Descrizione: Viene posto ad 1 il bit specificato della locazione determinata da s. s è definito nella descrizione delle analoghe istruzioni BIT.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: SET b, r

CB— b:	r: A	B	C	D	E	H	L
0	C7	C0	C1	C2	C3	C4	C5
1	CF	C3	C9	CA	CB	CC	CD
2	D7	D0	D1	D2	D3	D4	D5
3	DF	D8	D9	DA	DB	DC	DD
4	E7	E0	E1	E2	E3	E4	E5
5	EF	E8	E9	EA	EB	EC	ED
6	F7	F0	F1	F2	F3	F4	F5
7	FF	F8	F9	FA	FB	FC	FD

SET b, (HL)

SET b, (IX + d)

SET b, (IY + d)

b:	0	1	2	3	4	5	6	7
	C6	CE	Db	DE	E6	EE	F6	FE

Flag:

S	Z		H		P/V	N	C

(nessun effetto)

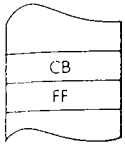
Esempio: SET 7, A

Prima:

A [61]

Dopo:

A [E1]

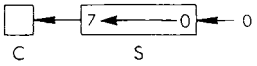


CODICE OGGETTO

SLA s

Spostamento aritmetico a sinistra dell'operando s.

Funzione:



Formato: s:

r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	0	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	0	1	1	0	byte 2: 26
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	1	0	0	1	1	0	byte 4: 26
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	1	0	0	1	1	0	byte 4: 26

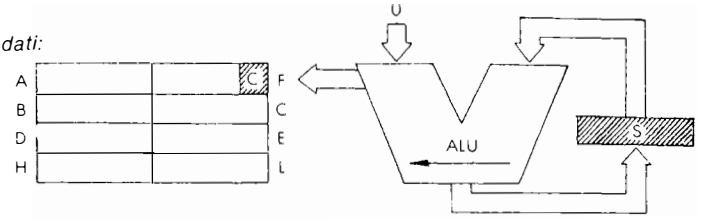
r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione:

Il contenuto della locazione determinata dall'operando specifico, è aritmeticamente spostato a sinistra con il contenuto del bit 7 che è spostato al flag di riporto ed uno 0 (zero) viene forzato nel bit 0. Il risultato finale è immagazzinato di nuovo nella locazione originale. s è definito nella descrizione delle analoghe istruzioni RLC.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	µsec @2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: SLA r

r:	A	B	C	D	E	H	L
CB	27	20	21	22	23	24	25

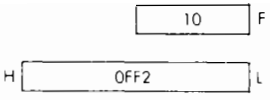
Flag:

S	Z	H	P/V	N	C
●	●		○	●	○	●

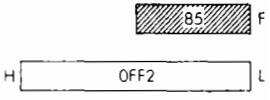
C è posto ad uno dal bit 7 del dato di sorgente.

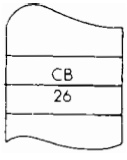
Esempio: SLA (HL)

Prima:

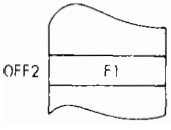


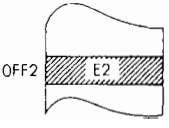
Dopo:





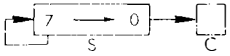
CODICE OGGETTO





SRA s Spostamento aritmetico a destra di s.

Funzione:



Formato:

s: r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	1	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	0	1	1	1	0	byte 2: 2E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	1	0	1	1	1	0	byte 4: 2E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	1	0	1	1	1	0	byte 4: 2E

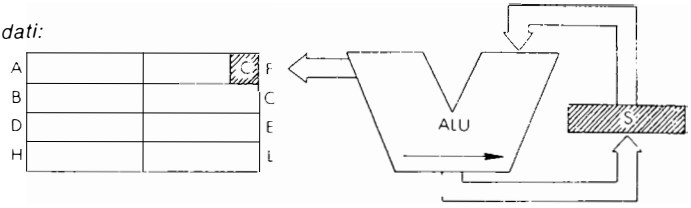
r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione:

Il contenuto della locazione determinata dall'operando specifico, è aritmeticamente spostato a destra. Il contenuto del bit 0 è mosso al flag di riporto ed il contenuto del bit 7 rimane invariato. Il risultato finale è immagazzinato nella locazione originale. s è definito nella descrizione delle analoghe istruzioni RLC.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	µsec @ 2 MHz:
r:	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: SRA r

r:	A	B	C	D	E	H	L
CB-	2F	28	29	2A	2B	2C	2D

Flag:

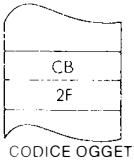
S	Z	H	P/V	N	C
●	●		○	●	○	●

C è posto ad uno dal bit 0 del dato di sorgente.

Esempio: SRA A

Prima:

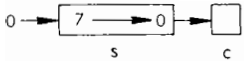
Dopo:



CODICE OGGETTO

SRL s Spostamento logico a destra di s.

Funzione:



Formato:

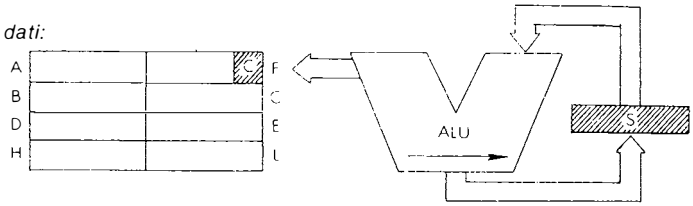
s:									
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	1	1	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	1	1	1	1	0	byte 2: 3E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	1	1	1	1	1	0	byte 4: 3E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valore di offset
	0	0	1	1	1	1	1	0	byte 4: 3E

r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: Il contenuto della locazione determinata dall'operando specifico, è logicamente spostato a destra. È mosso uno zero nel bit 7 ed il contenuto del bit 0 è mosso nel flag di riporto. Il risultato finale è immagazzinato di nuovo nella locazione originale.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Modo d'indirizzamento: r: implicito; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: SRL r

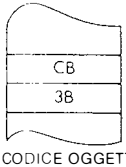
r	A	B	C	D	E	H	L
CB	3F	38	39	3A	3B	3C	3D

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	●

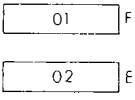
C è posto ad uno dal bit 0 del dato di sorgente.

Esempio: SRL E

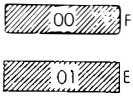


CODICE OGGETTO

Prima:



Dopo:



SUB A, s Sottrai l'operando s dall'accumulatore.

Funzione: A ← A − s

Formato: s: può essere r, n, (HL), (IX + d) oppure (IY + d)

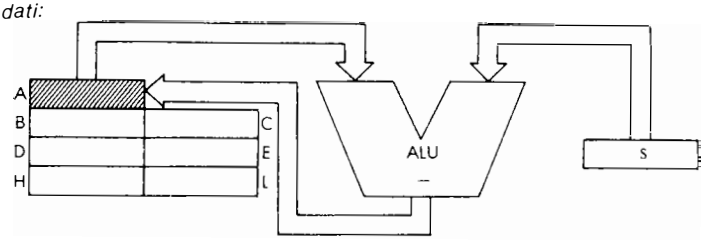
r	1	0	0	1	0	← r →				
n	1	1	0	1	0	1	1	0	byte 1: D6	
	← n →								byte 2: dato immediato	
(HL)	1	0	0	1	0	1	1	0	96	
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD	
	1	0	0	1	0	1	1	0	byte 2: 96	
	← d →								byte 3: valore di offset	
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD	
	1	0	0	1	0	1	1	0	byte 2: 96	
	← d →								byte 3: valore di offset	

r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: L'operando specificato s è sottratto dall'accumulatore ed il risultato è immagazzinato nell'accumulatore stesso. L'operando s è definito nelle descrizioni delle analoghe istruzioni ADD.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IX + d)	5	19	9.5

Modo d'indirizzamento: r: implicito; n: immediato; (HL): indiretto (IX + d), (IY + d): indicizzato.

Codici del byte: SUB A, r

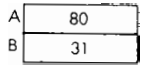
A	B	C	D	E	H	L
97	90	91	92	93	94	95

Flag:

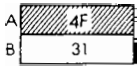
S	Z		H		P/V	N	C
●	●		●		●	1	●

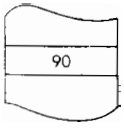
Esempio: SUB A, B

Prima:



Dopo:





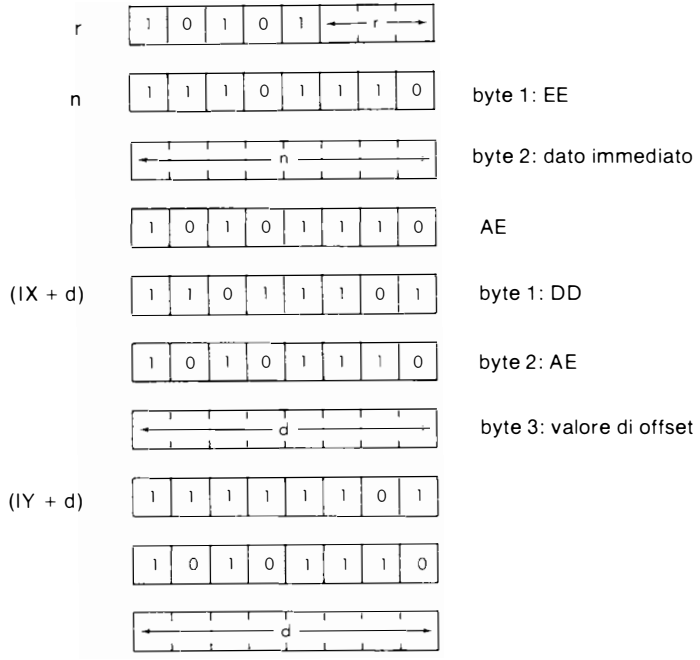
CODICE OGGETTO

XOR s

OR esclusivo tra accumulatore ed s.

Funzione: A ← A ∨ s

Formato: s: può essere r, n, (HL), (IX + d), oppure (IY + d)

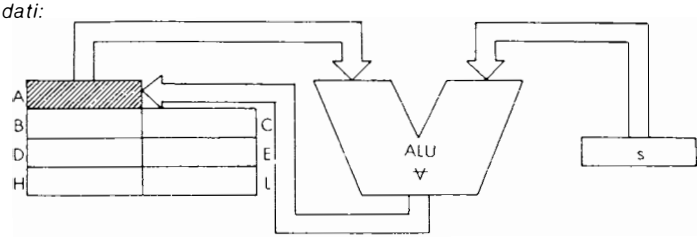


r può essere ognuno di questi:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descrizione: Viene eseguito l'OR esclusivo dell'accumulatore e dell'operando specificato s ed il risultato è immagazzinato nell'accumulatore. s è definito nella descrizione delle analoghe istruzioni ADD.

Flusso dei dati:



Temporizzazione:

s:	*cicli M:*	*stati T:*	*µsec @2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Modi d'indirizzamento: r: implicito; n: immediato; (HL): indiretto; (IX + d), (IY + d): indicizzato.

Codici del byte: XOR r:

A	B	C	D	E	H	L
AF	A8	A9	AA	AB	AC	AD

Flag:

S	Z		H		P/V	N	C
●	●		○		●	○	○

Esempio: XOR A, B1H

Prima:

A 36

Dopo:

A 87

CAPITOLO 5

TECNICHE DI INDIRIZZAMENTO

INTRODUZIONE

Questo capitolo presenterà la teoria generale dell'indirizzamento e le varie tecniche che sono state sviluppate per facilitare il recupero dei dati. In un secondo paragrafo saranno riesaminati i modi di indirizzamento specifici disponibili nello Z80, insieme con i loro vantaggi e limitazioni. Alla fine, per familiarizzare il lettore con i vari compromessi possibili, una sezione di applicazioni dimostrerà i possibili compromessi tra le varie tecniche di indirizzamento tramite lo studio di specifici programmi di applicazione.

Poiché lo Z80 ha diversi registri a 16 bit, oltre al contatore di programma, che possono essere usati per specificare un indirizzo, è importante che chi usa lo Z80 comprenda i vari modi d'indirizzamento, ed in particolare, l'uso dei registri indice.

Allo stato iniziale possono essere omessi i complessi modi di reperimento. Comunque, tutti i modi di indirizzamento sono utili per sviluppare programmi per questo microprocessore. Ora studiamo le varie alternative disponibili.

MODI POSSIBILI DI INDIRIZZAMENTO

L'*indirizzamento* si riferisce alla specifica entro una istruzione, della posizione dell'operando sul quale opererà l'istruzione. Adesso saranno esaminati i principali metodi di indirizzamento. Sono tutti illustrati nella Figura 5-1.

Indirizzamento implicito (o "Implicate" o "Registro")

Le istruzioni che operano esclusivamente sui registri usano normalmente l'*indirizzamento implicito*. Questo è illustrato nella Figura 5-1.

Una istruzione implicita deriva il suo nome dal fatto che non contiene in modo specifico l'indirizzo dell'operando sul quale opera. Invece, il suo codice operativo specifica uno o più registri, di solito l'accumulatore, oppure ogni altro registro o registri.

Siccome i registri interni sono di solito pochi in numero (comunemente otto), questo richiederà un piccolo numero di bit. Come esempio, tre bit all'interno dell'istruzione punteranno ad uno degli otto registri interni. Perciò, tali istruzioni possono normalmente essere codificate entro 8 bit. Questo è un vantaggio importante, poiché una istruzione ad otto bit normalmente si esegue più velocemente di qualsiasi istruzione di due o tre byte.

Un esempio di una istruzione implicita è:

```
LD A, B
```

che specifica "trasferisci il contenuto di B in A" (carica A da B).

Indirizzamento immediato

Nella Figura 5-1 è illustrato l'indirizzamento immediato. Il codice operativo ad otto bit è seguito da un literal ad 8 oppure 16 bit (una costante). Questo tipo di istruzione è necessaria, per esempio, per caricare un valore ad otto bit in un registro ad otto bit. Poiché il microprocessore è equipaggiato con registri a 16 bit, può anche essere necessario caricare literal a 16 bit.

Un esempio di una istruzione immediata è:

ADD A, 0H

La seconda parola di questa istruzione contiene il literal "0", che è aggiunto all'accumulatore.

Indirizzamento assoluto

L'indirizzamento assoluto di solito si riferisce al modo in cui i dati sono reperiti dalla memoria o posti nella memoria, in cui un codice operativo è seguito da un indirizzo a 16 bit. Perciò, l'indirizzamento assoluto richiede istruzioni di tre byte. Un esempio di indirizzamento assoluto è:

LD (1234 H), A

Specifica che il contenuto dell'accumulatore deve essere immagazzinato alla posizione di memoria esadecimale "1234".

Lo svantaggio dell'indirizzamento assoluto è quello di richiedere una istruzione di tre byte. Per migliorare l'efficienza del microprocessore, può essere disponibile un altro modo di indirizzamento con il quale per l'indirizzo è usata solo una parola: indirizzamento diretto.

Indirizzamento diretto (o "Breve" o "Relativo")

In questo modo di indirizzamento, il codice operativo è seguito da un indirizzo ad 8 bit. Anche questo è illustrato nella Figura 5-1.

Il vantaggio di questo approccio è quello di richiedere solo due byte invece di tre per l'indirizzamento assoluto. Lo svantaggio è quello di limitare tutto l'indirizzamento entro il campo di indirizzi da 0 a 255 oppure −128 a +127.

Quando si usa da 0 a 255 ("pagina zero"), questo è anche chiamato indirizzamento breve, o indirizzamento della pagina 0. Ogni qual volta è disponibile l'indirizzamento breve, per contrasto l'indirizzamento assoluto è spesso chiamato *indirizzamento esteso*.

La gamma da −128 a +127 è usata con le istruzioni di ramificazione. Questo è chiamato indirizzamento relativo.

Indirizzamento relativo

Le normali istruzioni jump (salto) o branch (ramificazione) richiedono otto bit per il codice operativo, più l'indirizzo a 16 bit al quale deve saltare il programma. Proprio come nel precedente esempio, questo modo ha lo svantaggio di richiedere tre parole, cioè, tre cicli di memoria. L'*indirizzamento relativo* usa solo un formato di due parole per fornire una ramificazione più efficiente. La prima parola è la descrizione dettagliata della ramificazione, di solito con il test che bisogna eseguire. La seconda parola è uno "spostamento".

Poiché lo spostamento deve essere positivo o negativo, una istruzione di ramificazione relativa permette una ramificazione in avanti fino a 127 posizioni (sette bit) oppure una ramificazione indietro fino a 128 posizioni (di solito +129 oppure −126, dal momento in cui PC sarà stato aumentato di due). Siccome la maggior parte dei loop tende ad essere corta, l'indirizzamento relativo può essere usato nella maggioranza dei casi e determina una esecuzione significativamente migliorata per tali programmi brevi. Come esempio, abbiamo già usato l'istru-

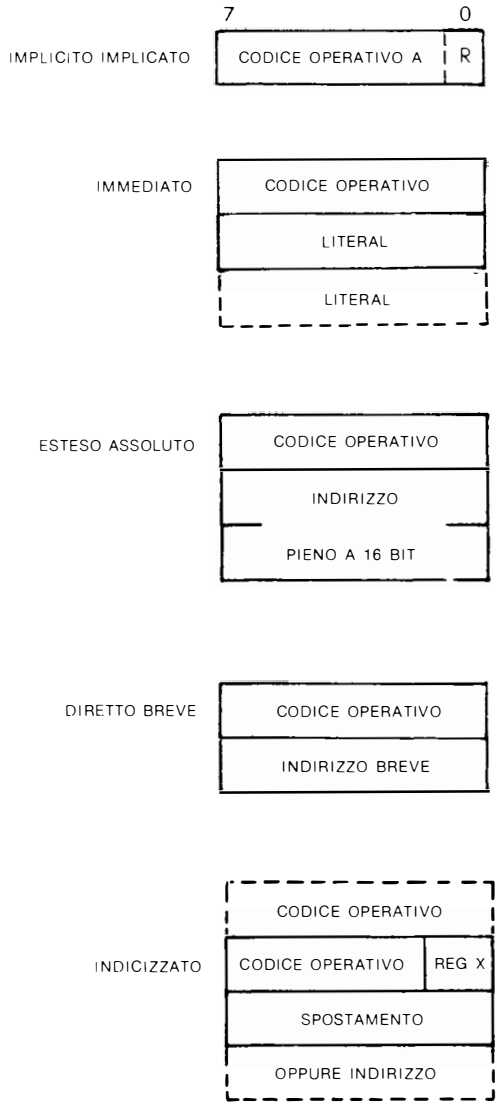


Figura 5-1: Modi d'indirizzamento di base

zione JR NC, la quale determina un "jump se non c'è nessun carry" ad una posizione entro 127 parole dall'istruzione branch (più precisamente +129 a −126).

I due vantaggi dell'indirizzamento relativo sono la migliorata esecuzione (meno byte usati, più alta velocità) e il riposizionamento del programma (indipendenza dagli indirizzi assoluti).

Indirizzamento indicizzato

L'indirizzamento indicizzato è la tecnica usata per accedere successivamente gli elementi di un blocco o di una tabella. Questo sarà illustrato con esempi più avanti in questo capitolo. Il principio dell'indirizzamento indiricizzato è quello che l'istruzione specifichi sia un registro indice che un indirizzo. Il contenuto del registro è aggiunto all'indirizzo per fornire l'indirizzo finale. In questo modo, l'indirizzo potrebbe essere l'inizio di una tabella posizionata nella memoria.

Il registro di indice sarebbe allora usato per accedere successivamente a tutti gli elementi di una tabella in un modo efficiente. (Questo richiede la disponibilità di istruzioni di incremento/decremento per il registro indice). In pratica, spesso esistono delle restrizioni che possono limitare la dimensione del registro indice, o la dimensione dell'indirizzo o del campo dello spostamento.

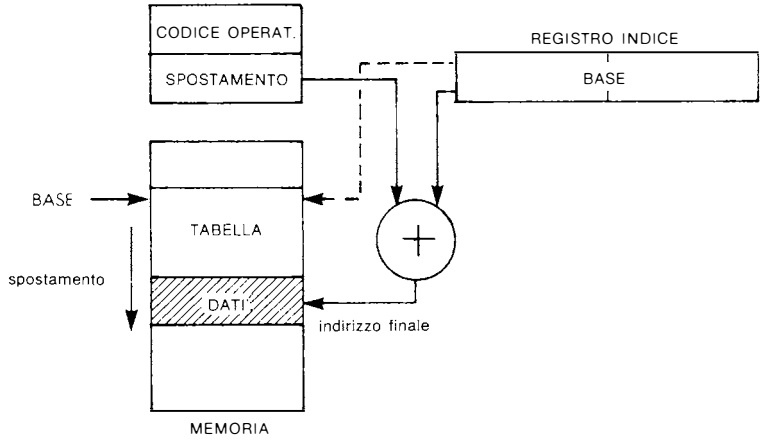


Figura 5-2: Indirizzamento (Pre-indicizzazione)

Pre-indicizzazione e post-indicizzazione

Possono essere distinti due modi di indicizzazione. La pre-indicizzazione è il solito modo di indicizzazione in cui l'indirizzo finale è la somma di uno spostamento o indirizzo e del contenuto del registro indice.

Questo è mostrato nella Figura 5-2 supponendo un campo di spostamento ad otto bit ed un registro indice a 16 bit.

La post-indicizzazione tratta il contenuto del campo di spostamento come l'*indirizzo* del vero spostamento, piuttosto che come spostamento. Questo è illustrato nella Figura 5-3. Nella post-indicizzazione, l'indirizzo finale è la somma del contenuto del registro indice più il contenuto della parola memoria *designata dal campo di spostamento.* Praticamente, questa caratteristica utilizza una combinazione dell'indirizzamento indiretto e della pre-indicizzazione. Ma non abbiamo ancora definito l'indirizzamento indiretto. Facciamolo ora.

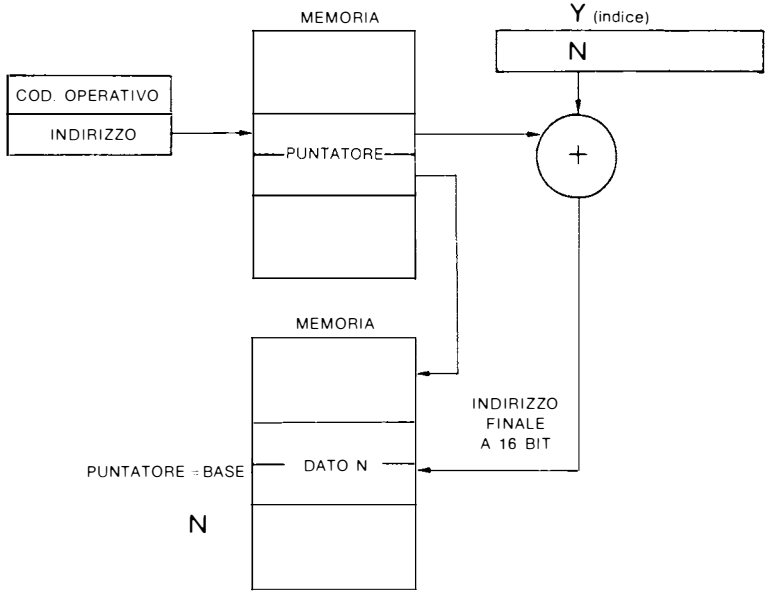


Figura 5-3: Indirizzamento indicizzato indiretto (Post-indicizzazione)

Indirizzamento indiretto

Abbiamo già visto che due sottoprogrammi possono voler scambiare una grande quantità di dati immagazzinati nella memoria. Più generalmente, diversi programmi, o diverse subroutine, possono aver bisogno di accedere ad un blocco comune d'informazioni. Per conservare la generalità del programma, è desiderabile non tenere un tale blocco ad una posizione di memoria fissa. In particolare, la dimensione di questo blocco potrebbe ingrossarsi o rimpicciolirsi, e può dover risiedere in varie aree della memoria, a seconda della sua dimensione. Perciò, non sarebbe pratico cercare di accedere a questo blocco usando indirizzi assoluti senza riscrivere il programma ogni volta.

La soluzione a questo problema sta nel depositare l'indirizzo di inizio del blocco ad una locazione di memoria fissa. Questo è analogo ad una situazione in cui diverse persone hanno bisogno di entrare in una casa, ed esiste solo una chiave. Con un accordo, la chiave della casa sarà nascosta sotto lo zerbino. Ogni abitante saprà allora dove guardare (sotto lo zerbino) per trovare la chiave di casa (o, forse, per trovare l'indirizzo comune di scheduling, per proporre una più stretta analogia). L'indirizzamento indiretto, perciò, usa normalmente un codice operativo (16 bit nel caso dello Z80) seguito da un indirizzo a 16 bit. Questo indirizzo è usato per recuperare una parola dalla memoria. Di solito, sarà una parola a 16 bit (nel nostro caso, due byte) dentro la memoria poiché è un indirizzo. Questo è illustrato dalla Figura 5-4. I due byte all'indirizzo specificato A1 contengono "A2". A2 è allora interpretato come il vero indirizzo dei dati ai quali si desidera aver accesso.

L'indirizzamento indiretto è particolarmente utile tutte le volte che sono usati i puntatori. Allora, varie aree del programma possono riferirsi a questi puntatori per aver accesso ad una parola o un blocco di dati in modo conveniente ed elegante. L'indirizzo finale può anche essere

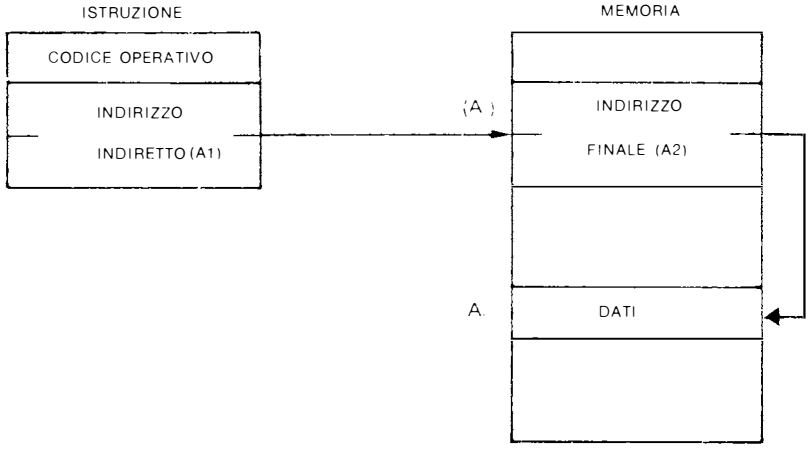


Figura 5-4: Indirizzamento indiretto

ottenuto puntando attraverso l'istruzione ad un registro a 16 bit in cui è contenuto l'indirizzo. Questo è chiamato "registro indiretto".

Combinazioni di modi

I modi di indirizzamento detti sopra possono essere combinati, in particolare, in uno schema d'indirizzamento completamente generale dovrebbe essere possibile usare molti livelli di indirizzamenti indiretti.

L'indirizzo A2 potrebbe essere interpretato di nuovo come un indirizzo indiretto e così via.

L'indirizzamento indicizzato può anche essere combinato con l'accesso indiretto. Ciò permette l'efficiente accesso alla parola n di un blocco di dati, a patto che si sappia dove sia il puntatore all'indirizzo d'inizio (vedere Figura 5-2).

Abbiamo acquisito familiarità con tutti i soliti modi di indirizzamento che possono essere forniti in un sistema.

La maggior parte dei sistemi di microprocessori, a causa della limitazione sulla complessità di una MPU, che deve essere realizzata all'interno di un singolo chip, non fornisce tutti i modi paossibili ma solo un piccolo sottoinsieme di questi.

Lo Z80 fornisce un buon sottinsieme di possibilità. Esaminiamole.

MODI DI INDIRIZZAMENTO DELLO Z80

Indirizzamento implicito (Z80)

L'indirizzamento implicito è essenzialmente usato dalle istruzioni a singolo byte che operano sui registri interni. Richiedono solo un ciclo per essere eseguite, poiché le istruzioni implicite operano esclusivamente sui registri interni.

Esempi di istruzioni che usano indirizzamento (o "registri") implicito sono: LDr, r'; ADD A, r; ADC A, s; SUB s; SBC A, s; AND s; OR s; XOR s; CP s; INC r.

Zilog inoltre distingue tra "indirizzamento di registro" e "indirizzamento implicito". L'indirizzamento implicito è allora limitato, in quella definizione, alle istruzioni che non hanno un campo specifico per puntare ad un registro interno. Questo introduce un altro modo di indirizzamento. Questa è una ragione perché il numero dei modi di indirizzamento è insufficiente a caratterizzare le capacità di un microprocessore.

Indirizzamento immediato

Poichè lo Z80 ha sia registri a lunghezza singola (8 bit), che coppie di registri a doppia lunghezza (16 bit), fornisce due tipi di indirizzamento immediato, sia con i literal da 8 bit che da 16 bit. Allora le istruzioni sono lunghe due o tre byte. Il primo byte contiene la costante, o literal, che deve essere caricata in un registro o usata per un'operazione.

LD IX e LD IY sono eccezioni che richiedono codici operativi a 16 bit.

Esempi di istruzioni che usano il modo di indirizzamento immediato sono:

LD r, n (due byte)
LD dd, nn (tre bite)

ed

ADD A, n (due byte)

Quando il literal è lungo due byte, il modo è chiamato "immediato esteso", nel caso dello Z80.

Indirizzamento assoluto o "esteso" (Z80)

Per definizione, l'indirizzamento richiede tre byte. Il primo byte è il codice operativo ed i due byte successivi sono l'indirizzo a 16 bit che specifica la locazione di memoria (l'"indirizzo assoluto"). In contrasto con l'"indirizzamento breve" (indirizzo ad 8 bit), questo modo è chiamato anche "indirizzamento esteso".

Esempi di istruzioni che usano l'indirizzamento esteso sono:

LD HL, (nn) e JP nn

dove nn rappresenta l'indirizzo della memoria a 16 bit, ed (nn) rappresenta il contenuto della locazione specificata.

Indirizzamento modificato in pagina zero (Z80)

L'indirizzamento in pagina zero non è disponibile nello Z80, eccetto attraverso l'istruzione CALL. Lo speciale modo di indirizzamento usato da questa istruzione è chiamata "indirizzamento modificato in pagina zero".

L'istruzione CALL contiene un campo di tre bit nelle posizioni bit b_5 b_4 b_3 usate per puntare ad una delle otto posizioni nella pagina zero della memoria.

L'indirizzo effettivo è b_5 b_4 b_3 000 ed è caricato nel PC. Poichè richiede solo un byte singolo, questa istruzione si esegue rapidamente, ed è facilmente generata nell'hardware. Era generalmente usata per rispondere a interrupt multipli (fino ad 8). Il suo svantaggio è quello di limitare la sequenza di esecuzione a 16 posizioni, oppure quello di richiedere un jump eliminando il vantaggio della velocità.

Questo perchè ciascun indirizzo a cui si può pervenire in pagina zero è lontano 16 byte dal successivo.

Questa istruzione è usata meno frequentemente adesso che i chip che controllano la priorità degli interrupt (PIC) sono diventati disponibili (vedere i riferimenti C201 e C207 per una descrizione dettagliata dei PIC). Un PIC farà uscire automaticamente un jump assoluto da 3 byte in risposta ad un riconoscimento di interrupt.

Questa istruzione è ora generalmente usata come un restart (rimessa in marcia).

Indirizzamento relativo (Z80)

Per definizione, l'indirizzamento richiede due byte. Il primo è il codice operativo "relativo al jump", mentre, il secondo byte specifica lo spostamento e il suo segno.

Per differenziare questo modo dall'istruzione di jump assoluto, viene indicato "JR".

Da un punto di vista della temporizzazione, questa istruzione dovrebbe essere esaminata con cauzione. Ogni qual volta un test, non è soddisfatto cioè, ogni qual volta non c'è nessuna ramificazione, questa istruzione richiede solo sette "cicli T". Questo perché la prossima istruzione che deve essere eseguita è già puntata dal program counter.

Comunque, quando il test è soddisfatto, cioè, ogni qual volta avviene il salto, questa istruzione richiede 12 "stati T"; un nuovo indirizzo effettivo deve essere calcolato e caricato nel contatore di programma.

Si deve prestare molta attenzione quando si calcola la durata di esecuzione di un segmento di programma. Ogni qual volta non si è sicuri se il salto avverrà o no, si deve prendere in considerazione il fatto che a volte il salto *richiederà 12 stati T, (condizione verificata) ed a volte 7 (condizione non verificata).*

Quando si progetta un loop, l'esecuzione sarà perciò più veloce usando un JR (jump Relativo) che prova una condizione che di solito *non è verificata*, come una condizione di "non zero" per il contatore.

Quando i JR sono usati al di fuori dei loop, e non è nota la condizione sotto test, per la durata di JR è spesso usato un valore di temporizzazione medio.

Questo problema della temporizzazione non si applica al jump incondizionato JR e. Infatti esso non prova nessuna condizione e dura sempre 12 stati T.

Indirizzamento indicizzato (Z80)

Questo modo di indirizzamento non esisteva nell'8080, e fu aggiunto allo Z80 (così come i due registri indice). Come risultato, divenne necessario aggiungere un byte extra al codice operativo portandolo a 16 bit nel set di istruzioni dello Z80 (LDIR è un altro esempio di codice operativo a 16 bit). Nella Figura 5-5 è mostrata la struttura di un'istruzione indicizzata.

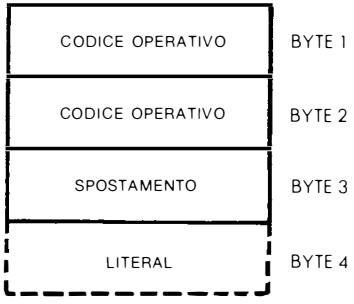


Figura 5-5: L'indirizzamento indicizzato ha un codice operativo di 2 byte

Le istruzioni che permettono l'indirizzamento indicizzato sono:

LD, ADD, INC, RLC, BIT, SET.

Questo modo sarà usato estesamente nei programmi che operano su blocchi di dati, tabelle o liste.

Indirizzamento indiretto (Z80)

Lo Z80 fornisce una capacità limitata di indirizzamento indiretto chiamata "Indirizzamento Indiretto di Registro" ("Register Indirect Addressing"). In questo modo ognuna delle coppie di registri a 16 bit BC, DE, HL può essere usata come un indirizzo di memoria.

Ogni qual volta essi puntano ad un dato di 16 bit, puntano la parte più bassa. La parte più alta risiede nell'indirizzo sequenzialmente successivo (maggiore).

Combinazioni di modi

Le combinazioni di modi sono essenzialmente non esistenti, eccetto che le istruzioni che si riferiscono ai due operandi possono usare un tipo differente di indirizzamento per ognuno.

Perciò, una istruzione di *carico* o aritmetica può accedere ad un operando nel modo immediato, e l'altra attraverso un accesso indicizzato.

Poi, il meccanismo d'indirizzamento di bit può accedere all'ottavo bit del byte attraverso uno dei tre modi d'indirizzamento, come spiegato nel paragrafo precedente.

Nelle tabelle del precedente capitolo sono indicati i modi d'indirizzamento specifico disponibili per ogni istruzione.

Indirizzamento di bit

L'indirizzamento del bit non è generalmente considerato un modo d'indirizzamento se l'indirizzamento è definito come accedere ad un *byte*. Comunque, definito come un modo oppure un gruppo d'istruzioni, è un servizio prezioso.

Dal momento in cui è definito come un "modo d'indirizzamento" nella nomenclatura Zilog, qui sarà descritto in questa maniera. È specifico dello Z80 e non era fornito sullo 8080. L'indirizzamento di bit si riferisce al meccanismo di accesso a bit specificati. Lo Z80 è fornito di speciali istruzioni per porre ad 1, a 0 resettare o provare specificati bit di una locazione di memoria o di un registro. Si può accedere al bit specificato attraverso uno dei tre modi d'indirizzamento: registro, registro indiretto, ed indicizzato. Sono usati tre bit all'interno del codice operativo per scegliere un bit su otto.

IMPIEGO DEI MODI DI INDIRIZZAMENTO DELLO Z80

Indirizzamento lungo e breve

Abbiamo già usato le istruzioni di jump relativo in vari programmi che abbiamo già sviluppato. Esse si spiegano da sole. Una domanda interessante è: Cosa possiamo fare se il range ammissibile per la ramificazione non è sufficiente per i nostri scopi? Una soluzione semplice è quella di usare un cosiddetto *jump lungo*. Questo è semplicemente un salto da una posizione che contiene una specificazione assoluta o jump "lungo":

```
JR NC, $+3          SALTA ALL'INDIRIZZO CORRENTE + 3 SE C'È ZERO
JP FAR              ALTRIMENTI SALTA A FAR
(ISTRUZIONE SUCCESSIVA)
```

Il precedente programma di due righe porta al salto alla posizione FAR ogni qual volta il carry è uguale ad 1.

Questo risolve il problema di jump lungo. Perciò, ora consideriamo i modi d'indirizzamento più complessi, cioé, indicizzazione ed indirezione.

Impiego dell'indicizzazione per gli accessi a blocchi sequenziali

L'indicizzazione è principalmente usata per indirizzare posizioni successive entro una tabella. La restrizione è che la lunghezza massima deve essere inferiore a 256 in modo che lo spostamento possa risiedere in un registro indice ad otto bit. Abbiamo imparato a controllare un carattere. Ora cercheremo la presenza di un "*" in una tabella di 100 elementi. L'indirizzo di i

nizio per questa tabella è chiamato BASE. La tabella ha soltanto 100 elementi. Il programma è il seguente (vedere il diagramma di flusso nella Figura 5-6):

```
SEARCH    LD    IX, BASE
          LD    A, "*"
          LD    B, COUNT
TEST      CP    (IX)
          JR    Z, FOUND
          INC   IX
          DEC   B
          JR    NZ, TEST
NOTFND
```

Nel paragrafo successivo relativo al Trasferimento di Blocchi, sarà presentato un programma migliorato.

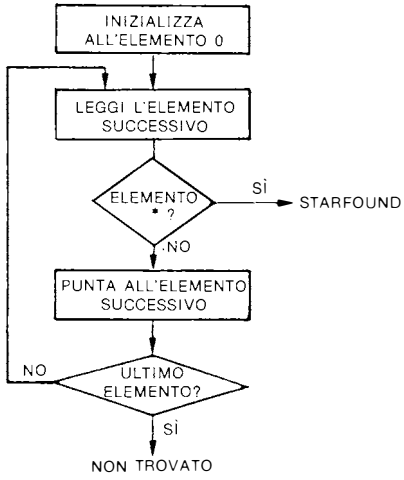


Figura 5-6: Diagramma di flusso per la ricerca di un carattere

Un programma per il trasferimento di blocchi con meno di 256 elementi

Chiameremo "COUNT" il numero di elementi che deve essere mosso nel blocco. Si presuppone che il numero sia inferiore a 256. FROM è l'indirizzo di base del blocco. TO è la base dell'area di memoria dove dovrebbe essere mosso. L'algoritmo è abbastanza semplice: muoveremo una parola alla volta, tenendo presente a quale parola stiamo muovendo, immagazzinando la sua posizione nel contatore C. Il programma è il seguente:

```
BLKMOV    LD    IX, FROM
          LD    IY, TO
          LD    C, COUNT
NEXT      LD    A, (IX)        ACCETTA PAROLA
          LD    (IY), A
          INC   IX
          INC   IY
          DEC   C
          JR    NZ, NEXT
```

Esaminiamolo:

```
BLKMOV    LD    IX, FROM
          LD    IY, TO
          LD    C, COUNT
```

Queste tre istruzioni inizializzano rispettivamente i registri IX, IY e C, come illustrato nella Figura 5-7. Il registro indice IX è usato come il puntatore di sorgente, e sarà regolarmente aumentato.

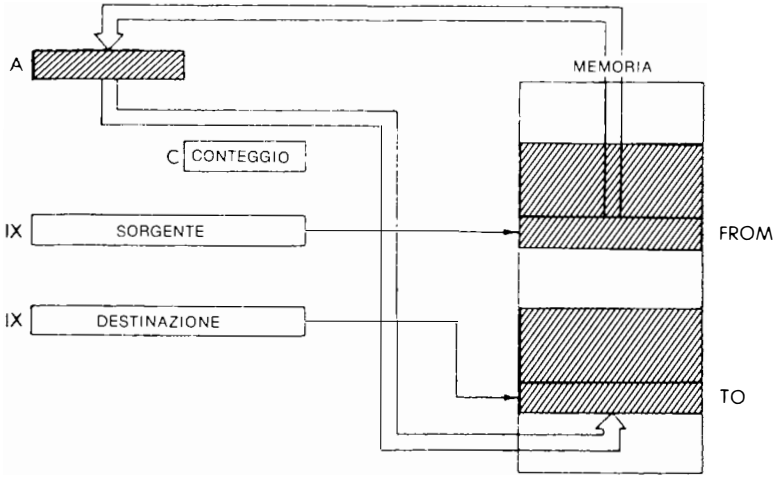


Figura 5-7: Trasferimento di blocchi: inizializzazione del registro

Il registro indice IY è usato come il puntatore di destinazione, e sarà incrementato regolarmente. Il registro C è caricato con il numero massimo di elementi che devono essere trasferiti (limitato a 256 poiché questo è un registro ad otto bit), e sarà regolarmente diminuito. Tutti gli elementi sono stati trasferiti ogni qual volta C arriva a zero.

Le prossime due istruzioni:

```
NEXT      LD    A, (IX)
          LD    (IY), A
```

Caricano il contenuto della locazione di memoria puntata da IX nell'accumulatore, poi lo trasferiscono nella locazione di memoria puntata dal registro IY. In altre parole, queste due istruzioni trasferiscono un elemento del blocco di sorgente nel blocco di destinazione. I due registri indice sono poi aumentati.

```
          INC   IX
          INC   IY
```

E il registro contatore è diminuito:

```
          DEC   C
```

Alla fine, fino a quando il contatore non è 0, il "loop" ritorna alla label NEXT:

```
          JR    NZ, NEXT
```

Questo è un esempio della possibile utilizzazione dei registri indice.

Comunque, confrontiamolo con lo stesso programma scritto per un altro microprocessore, il MOS Technology 6502, che è pure fornito di una capacità d'indicizzazione, ma usa convenzioni diverse (cioè, ha limitazioni differenti su una indicizzazione a scopo generale). Il programma è il seguente:

```
        LDX     #NUMBER
NEXT    LDA     FROM, X
        STA     TO, X
        DEX
        BNE     NEXT
```

Senza entrare nei dettagli del programma qui sopra, il lettore noterà immediatamente quanto è più corto del precedente.

Questo perché il registro indice X è usato come uno spostamento variabile, laddove BASE e TEST sono usate come gli indirizzi di sorgente fissa e di destinazione.

Questo esempio dovrebbe sottolineare che benché in teoria l'indicizzazione sia un servizio potente, non conduce necessariamente ad una efficiente codifica, a causa delle limitazioni d'indirizzamento imposte su di essa nel caso di vari microprocessori. Veramente l'indicizzazione a scopo generale richiede la possibilità di uno spostamento a 16 bit o campo d'indirizzo cosi come di un registro di indice a 16 bit. Comunque, dovrebbe essere noto che questo specifico problema è risolto, nello Z80 dalla presenza di istruzioni specializzate. Sarà ora descritto un trasferimento di blocchi a scopo generale che può essere compiuto in sole quattro istruzioni.

Comunque, per essere giusti verso lo Z80, suggeriamo esercizi addizionali per il lettore:

Esercizio 5-1: *Scrivete il programma per il trasferimento di blocchi per lo Z80 nello stile del programma di sopra per il 6502, cioè, presumendo che il registro indice contenga uno spostamento. Presumete che il blocco di sorgente e di destinazione siano localizzati nella pagina 0, cioè agli indirizzi da 0 a 256. Naturalmente, sarà presupposto che il numero di elementi entro ogni blocco sia abbastanza piccolo affinché non si sovrappongano.*

Esercizio 5-2: *Presumiamo adesso che i blocchi di sorgente e di destinazione siano situati in qualche parte della manovra, eccetto che siano entrambi entro la stessa pagina. In quel caso riscrivete il programma precedente.*

Programma per il trasferimento di blocchi generalizzato (più di 256 elementi)

L'assegnazione dei registri e la mappa della memoria sono mostrate nella Figura 5-8. Il programma è il seguente:

```
LD      BC, COUNT       NUMERO DI BYTE
LD      DE, TO          INDIRIZZO DI DESTINAZIONE
LD      HL, FROM        INDIRIZZO D'INIZIO
LDIR                    TRASFERISCI TUTTI I BYTE
```

Memoria usata: 11 byte
Temporizzazione: 21 cicli/byte

La prima istruzione è:

```
LD      BC, COUNT
```

Carica il numero di elementi che devono essere trasferiti (un valore a 16 bit) nella coppia di

registri BC. Le prossime due istruzioni inizializzano rispettivamente la coppia di registri DE e la coppia di registri HL:

```
LD      DE, TO
LD      HL, FROM
```

Alla fine, la quarta istruzione:

```
LDIR
```

esegue il trasferimento completo.

LDIR è un'istruzione per il *trasferimento automatico dei blocchi*. La sua potenza dovrebbe essere ovvia da questo esempio. LDIR agisce nella seguente sequenza:
Il contenuto della locazione di memoria puntata da H e L è trasferito nella locazione di memoria puntata da DE: (DE) = (HL). Poi, DE è aumentato: DE = DE + 1. Successivamente HL è aumentato: HL = HL + 1. Quindi BC è diminuito: BC = BC − 1. Se BC diventa 0, l'istruzione è terminata. Altrimenti l'istruzione è ripetuta.

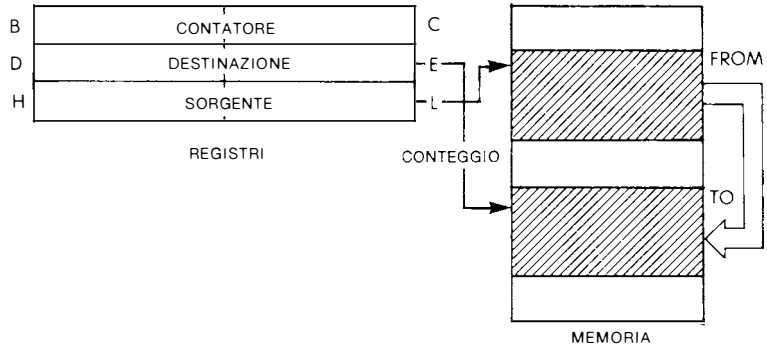


Figura 5-8: Mappa di memoria per il trasferimento di un blocco

Senza ulteriori commenti, a questo punto dovrebbe essere apparente il valore e la potenza dell'istruzione LDIR. Allo stesso modo, la nostra ricerca per il carattere "*" può essere migliorata dall'uso di un'istruzione automatica, CPIR, speciale per lo Z80. Il programma corrispondente è il seguente:

```
        LD      A,      '*
        LD      BC,     COUNT
        LD      HL,     STRING
STAR    CPIR
        JR      Z,      STAR
NOSTAR  ---
```

La prima istruzione carica l'accumulatore con il codice per il carattere asterisco (star). Quindi la coppia di registri BC è inizializzata per il conteggio del numero di parole che devono essere cercate all'interno del blocco:

```
        LD      BC,     COUNT
```

La coppia di registri H ed L è caricata con l'indirizzo di inizio del blocco che deve essere cercato (STRING).

Quindi è eseguita l'istruzione automatica:

```
LD,    HL, STRING
CPIR
```

L'istruzione CPIR è un'istruzione di confronto automatico. Il contenuto della locazione di memoria specificato dell'indirizzo contenuto in H ed L è confrontato con il contenuto dell'accumulatore.

Se il confronto è soddisfatto allora lo Z del registro dei flag sarà posto ad 1. Quindi la coppia di registri H ed L viene diminuita.

L'istruzione è ripetuta fino a quando la coppia BC va a 0 oppure il confronto è soddisfatto. Dopo che è eseguita l'istruzione CPIR, è così necessario provare il flag Z per determinare se è riuscito il confronto (il CPIR potrebbe al limite aver riciclato su 64 K parole senza successo). Questo è lo scopo dell'ultima istruzione del programma:

```
JR     Z, STAR
```

Esercizio 5-3: *Riscrivete il programma precedente in modo che la ricerca proceda all'indietro (Consiglio: Usate l'istruzione CPIR) "Continuate il trasferimento dei blocchi fino a quando è trovato l'asterisco".*

Adesso sviluppiamo un programma che unisca le caratteristiche dei due programmi precedenti.

Compieremo il trasferimento di blocchi dalla posizione FROM alla posizione TO, che si dovrà arrestare automaticamente ogni qual volta è trovato un carattere asterisco, lo "star". Il programma è il seguente:

```
        LD    BC, COUNT
        LD    HL, FROM
        LD    DE, TO
        LD    A, '*          DELIMITATORE
TEST    CP    (FROM)         CONFRONTA CON IL CARATTERE
                             IN MEMORIA
        JR    Z, END         TERMINA SE HA BUON ESITO
        LDI                  TRASFERISCI CARATTERE E AGGIORNA
                             PUNTATORI E CONTEGGIO
        JR    PE, TEST       CONTINUA A PROVARE A MENO
                             CHE SIA GIÀ FATTO
                             P INDICA SE BC = 0
```

Le prime tre istruzioni del programma eseguono la solita inizializzazione sistemando i registri del contatore ed i puntatori di sorgente e di destinazione:

```
LD     BC, COUNT
LD     HL, FROM
LD     DE, TO
```

Il carattere di star è depositato, "come al solito" nell'accumulatore, in modo che possa essere confrontato con il carattere letto dalla locazione di memoria

```
LD     A, '*
```

Questo è esattamente quello che è fatto dalla prossima istruzione:

```
TEST   CP (FROM)
```

Il successo o il fallimento del confronto è determinato provando il bit Z. Il bit Z sarà stato posto ad 1 se il confronto è soddisfatto. Questo è eseguito dalla prossima istruzione:

```
        JR      Z, END
```

La prossima istruzione è un'istruzione per il *trasferimento automatico*.

```
        LDI
```

Questa istruzione trasferisce il carattere, aggiorna i puntatori ed il contatore in una istruzione singola. LDI trasferisce il contenuto puntato da H ed L nella locazione di memoria puntata da D ed E: (DE) = (HL).
Inoltre aumenta DE ed HL:

DE := DE + 1
HL := HL + 1

Alla fine, diminuisce BC: BC diventa BC − 1. La particolarità di questa istruzione è quella che il flag P/V è posto ad 1 se BC diminuisce a "0" e posto a 0 in *caso contrario*.
Ciò sarà esplicitamente provato dall'ultima istruzione nel programma per determinare se si deve uscire.

```
        JR      PE, TEST
```

Somma di due blocchi

Qui sarà sviluppato un programma per sommare, elemento dopo elemento, due blocchi iniziando rispettivamente agli indirizzi BLK 1, BLK 2, ed avendo un ugual numero di elementi, COUNT. Il programma è il seguente:

```
BLKADD  LD      IX, BLK1
        LD      IY, BLK2
        LD      B, COUNT
        XOR     A
LOOP    LD      A, (IX + 0)
        ADC     A, (IY + 0)
        LD      (IX), A
        DEC     IX
        DEC     IY
        DEC     B
        JR      NZ, LOOP
```

La mappa della memoria è mostrata nella Figura 5-9. Il programma è facile. Il numero di elementi che deve essere sommato è caricato nel registro di contatore B, ed i due registri indice IX e IY sono inizializzati ai loro valori BLK1 e BLK2:

```
BLK ADD LD      IX, BLK1
        LD      IY, BLK2
        LD      B, COUNT
```

Il bit di carry è poi cancellato prima della prima addizione.

```
        XOR A
```

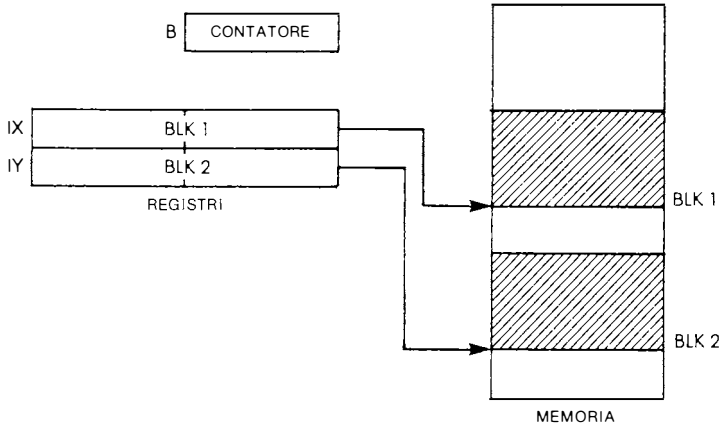


Figura 5-9: Addizione di due blocchi: BLK1 = BLK1 + BLK2

Il primo elemento è caricato nell'accumulatore:

```
LOOP    LD    A, (IX + 0)
```

Ad esso è poi sommato il corrispondente elemento di BLK2:

```
        ADC   A, (IY + 0)
```

e poi salvato nell'elemento di BLK1:

```
        LD    (IX), A
```

I due registri puntatori IX ed IY sono decrementati:

```
        DEC   IX
        DEC   IY
```

così come il registro contatore:

```
        DEC   B
```

Il loop dell'addizione è eseguito fino a quando il registro contatore non è 0:

```
        JR    NZ, LOOP
```

Esercizio 5-4: *Potete usare il programma precedente per eseguire un'addizione a 32 bit?*

Esercizio 5-5: *Potete usare il programma precedente per eseguire un'addizione a 64 bit?*

Esercizio 5-6: *Modificate il programma precedente in modo che il risultato sia immagazzinato in un blocco separato che inizi all'indirizzo BLK3.*

Esercizio 5-7: *Modificate il programma precedente per eseguire una sottrazione piuttosto che un'addizione.*

Esercizio 5-8: *Modificate il precedente programma originale in modo che BLK1 e BLK2 siano nella parte alta di ogni blocco piuttosto che nella parte bassa (vedere Figura 5-10).*

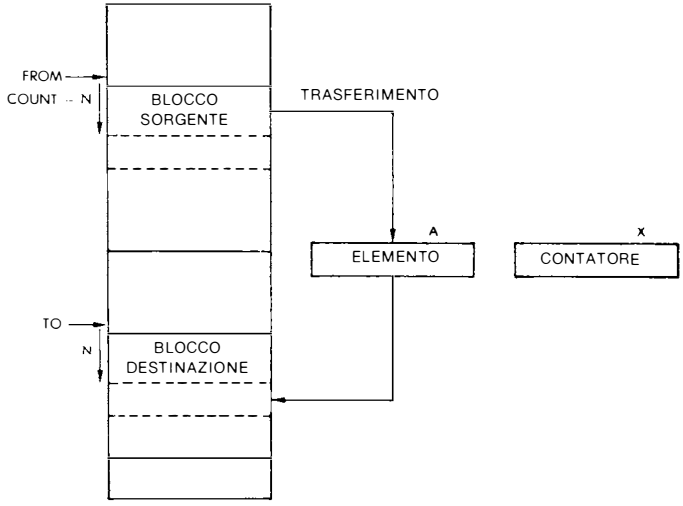


Figura 5-10: Organizzazione della memoria per il trasferimento di blocchi

SOMMARIO

È stata presentata una descrizione completa dei modi d'indirizzamento. È stato mostrato che lo Z80 offre molti meccanismi possibili e sono stati analizzati i modi d'indirizzamento specifici disponibili sullo Z80. Infine sono stati presentati diversi programmi applicativi per dimostrare il valore dei vari meccanismi d'indirizzamento. Per programmare efficientemente lo Z80, è richiesta una comprensione di questi meccanismi.

Saranno usati durante tutto il resto di questo libro.

ESERCIZI

5-9: *Scrivete un programma per sommare i primi 10 byte di una tabella immagazzinata alla posizione "BASE". Il risultato avrà 16 bit. (Questo è un calcolo "checksum" o somma di controllo).*

5-10: *Potete risolvere lo stesso problema senza usare il modo d'indicizzazione?*

5-11: *Invertite l'ordine dei 10 byte di questa tabella. Immagazzinate il risultato all'indirizzo "REVER".*

5-12: *Cercate nella stessa tabella l'elemento più grande. Immagazzinatelo all'indirizzo di memoria "LARGE".*

5-13: *Sommate insieme gli elementi corrispondenti di tre tabelle, le cui basi sono BASE 1, BASE 2, BASE 3. La lunghezza di queste tre tabelle è immagazzinata nella pagina zero all'indirizzo "LENGTH".*

CAPITOLO 6

TECNICHE DI INPUT/OUTPUT

INTRODUZIONE

Finora abbiamo imparato come scambiare informazioni tra la memoria e i vari registri del processore. Abbiamo imparato a manipolare i registri e ad usare una varietà d'istruzioni per manipolare i dati. Ora dobbiamo imparare a comunicare con il mondo esterno. Questo è chiamato input-output (ingresso/uscita). L'*input* si riferisce alla cattura di dati dalle unità periferiche esterne (tastiera, disk, o sensore fisico). L'*output* si riferisce al trasferimento di dati dal microprocessore o dalla memoria ai dispositivi esterni come una stampante, un CRT, un disk, oppure sensori e relé.

Procederemo in due fasi. Preliminarmente impareremo ad eseguire le operazioni di input/output richieste tramite dispositivi comuni.

Come seconda cosa, impareremo a gestire simultaneamente diversi dispositivi di input/output. cioé ad eseguire lo *scheduling*. Questa seconda parte coprirà, in particolare, il polling in funzione degli "interrupt".

INPUT/OUTPUT

In questo capitolo, impareremo a rivelare oppure a generare segnali semplici come gli impulsi. Poi studieremo le tecniche per forzare o misurare la temporizzazione corretta. Quindi saremo pronti per tipi più complessi di input/output, come i trasferimenti ad alta velocità in serie e paralleli.

Le istruzioni di input/output dello Z80

Lo Z80 è fornito di un set speciale d'istruzioni di input output. La maggior parte dei microprocessori ad otto bit non sono forniti di un set speciale di istruzioni di input/output ed usano il set generale di istruzioni sui dispositivi d'input/output. Lo Z80, come l'8080 è fornito di istruzioni d'input e d'output fondamentali. Comunque, lo Z80 è fornito anche d'istruzioni d'input/output addizionali. Queste saranno descritte qui più dettagliatamente per facilitare la comprensione dei programmi che saranno presentati nell'arco di questo capitolo.

Le istruzioni d'input e output fondamentali sono rispettivamente IN A, (n) ed OUT (n), A. Queste due istruzioni sono ereditate dall'8080. Scriveranno e leggeranno rispettivamente un byte tra la porta scelta e l'accumulatore. Il vero processo d'indirizzamento è tale che l'indirizzo "n" del dispositivo I/O è fornito sulle linee da A0 ad A7 del bus degli indirizzi, mentre il contenuto dell'accumulatore compare sulle linee di indirizzo da A8 ad A15. Quando sono indirizzati solo 256 dispositivi, può essere necessario azzerare il contenuto dell'accumulatore in modo esplicito se qualcuna delle linee d'indirizzo da A8 ad A15 può essere decodificata da un dispositivo di I/O. Negli esempi semplici che seguono, presumeremo che siano presenti meno di 256 dispositivi e che non siano collegati agli indirizzi da A8 ad A15, in modo che non sarà necessa-

rio azzerare esplicitamente il contenuto dell'accumulatore, per esempio prima d'usare l'istruzione IN.

Una istruzione d'input speciale: IN r, (C), permette l'uso del contenuto del registro C come indirizzo del dispositivo I/O. Quando si usa questa istruzione, il contenuto del registro B fornisce automaticamente la parte alta dell'indirizzo (da A8 ad A15). Il registro specificato r è caricato dall'indirizzo specificato.

"r" può essere uno dei soliti 7 registri general-purpose (per scopi generali).

Generazioni di un segnale

Nel caso più semplice, un dispositivo d'output sarà spento (o acceso) dal computer. Per cambiare lo stato del dispositivo di output, il programmatore cambierà soltanto livello da uno "0" logico ad un "1" logico oppure da "1" a "0". Presumiamo che un relé esterno sia collegato al bit "0" di un registro chiamato "OUT 1". Per accenderlo, noi scriveremo semplicemente un "1" nella giusta posizione bit del registro.

Presumiamo qui che OUT 1 rappresenti l'indirizzo di questo registro di output all'interno del nostro sistema. Un programma che commuta ON il relé è:

```
TURNON   LD   A, 00000001B   CARICA IL PATTERN IN A
         OUT  (OUT1), A      MANDALO AL DISPOSITIVO
```

dove OUT è l'istruzione di output.

Qui abbiamo presupposto che è irrilevante lo stato degli altri sette bit del registro. Comunque, questo non è il caso più frequente. Questi bit potrebbero essere collegati ad altri relé. Perciò, miglioriamo questo programma semplice.

Possiamo commutare ON il relé, senza variare lo stato di altri bit all'interno di questo registro.

Supporremo che sia possibile leggere e scrivere il contenuto di questo registro. Il nostro programma migliorato adesso diventa:

```
TURNON   IN   A, (OUT1)      LEGGI IL CONTENUTO DI OUT1
         OR   00000001B      FORZA IL BIT "0" AD "1" IN A
         OUT  (OUT1), A
```

Il programma per prima cosa legge il contenuto della posizione OUT1, poi esegue un OR inclusivo sul suo contenuto. Questo varia soltanto la posizione di bit da 0 ad "1" e lascia intatto il resto del registro. (Riferitevi al Capitolo 4 per ulteriori dettagli sull'operazione OR). Questo è illustrato dalla Figura 6-1.

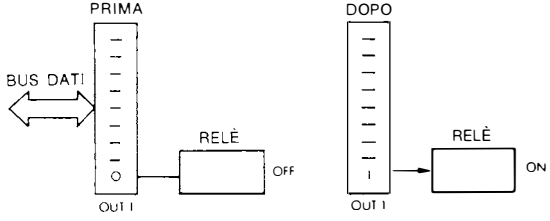


Figura 6-1: Accensione di un Relé

Impulsi

La generazione di un *impulso* è compiuta esattamente come nel caso del *livello* detto prima. Inizialmente un bit di output è commutato ON e poi OFF più tardi. Questo genera un impulso.

Ciò è illustrato nella Figura 6-2. Questa volta, comunque, deve essere risolto un problema addizionale: si deve generare l'impulso di lunghezza corretta. Perciò, studiamo la generazione di un ritardo calcolato.

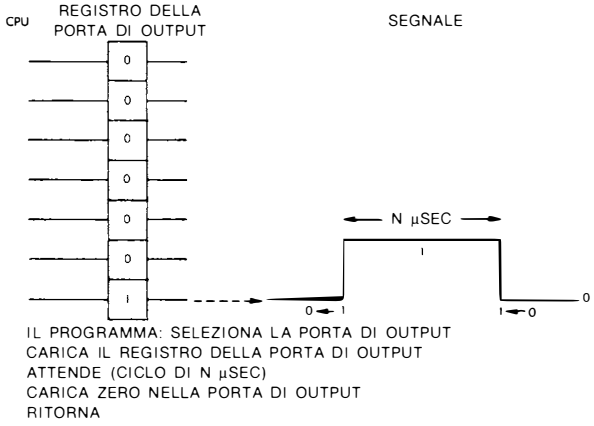


Figura 6-2: Impulso programmato

Generazione e misurazione del ritardo

Può essere generato un ritardo con metodi software ed hardware. Qui studieremo il modo di eseguirlo tramite programma, e più tardi mostreremo come può anche essere eseguito con un contatore hardware, chiamato timer ad intervallo programmabile (PIT).

I ritardi programmati sono ottenuti tramite conteggio. Un registro contatore è caricato con un valore, poi è decrementato. Il programma cicla su se stesso e continua a decrementare fino a quando il contatore raggiunge il valore "0". La lunghezza totale di tempo di questo processo realizzerà il ritardo richiesto. Come esempio, generiamo un ritardo di 67 cicli di clock

```
DELAY   LD   A, 5        A È IL CONTATORE
NEXT    DEC  A           DECREMENTA
        JP   NZ, NEXT    NEXT TEST
```

Questo programma carica A con il valore 5. La prossima istruzione decrementa A e l'istruzione seguente farà si che una ramificazione a NEXT avvenga fino a che A non decrementi a "0".

Quando finalmente A decrementa a zero, il programma uscirà da questo loop ed eseguirà qualsiasi istruzione successiva.

La logica del programma è semplice e compare nel diagramma di flusso della Figura 6-3.

Adesso calcoliamo il ritardo effettivo che sarà compiuto dal programma. Nell'Appendice del libro, guarderemo il numero di cicli richiesti da ognuna di queste istruzioni: LDA, nel modo immediato, richiede nove cicli di clock. DEC userà quattro cicli. Alla fine, JP userà sette cicli eccetto durante l'ultima iterazione, dove userà 12 cicli. Verificate che esistono due possibilità quando guardate nella tabella al numero di cicli per JP: se non avviene la ramificazione, JP richiederà solo sette cicli. Se la ramificazione avviene, che sarà il caso più comune durante il loop, allora sono richiesti 12 cicli.

Perciò, la temporizzazione è di sette cicli per la prima istruzione. più 11 cicli per le prossime

due, moltiplicato per il numero di volte che sarà eseguito il loop, più un ritardo extra di cinque cicli per l'ultimo JP non riuscito:

Ritardo = 7 + 11 x 5 + 5 = 67 cicli

Presumendo un ciclo di 0,5 microsecondi, il ritardo programmato sarà di 33,5 microsecondi.

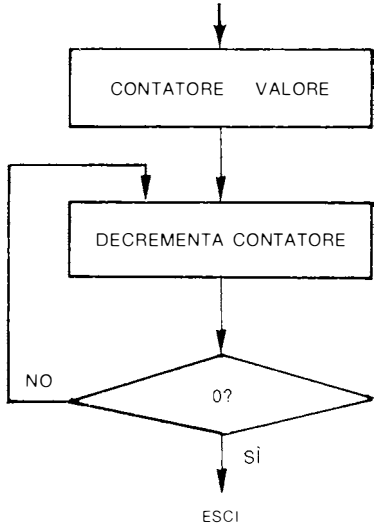


Figura 6-3: Diagramma di flusso del ritardo

Il loop di ritardo che è stato descritto è usato nella maggior parte dei programmi d'input/output. Dovrebbe essere ben compreso. Cercate di fare i seguenti esercizi.

Esercizio 6-1: *Quali sono i ritardi massimi e minimi che possono essere compiuti con queste tre istruzioni?*

Esercizio 6-2: *Modificate il programma per ottenere un ritardo di circa 100 microsecondi.*

Se qualcuno desidera realizzare un ritardo più lungo, una soluzione semplice è quella di aggiungere istruzioni extra nel programma, tra DEC e JP. Il modo più semplice per farlo è quello di aggiungere istruzioni NOP. (Il NOP non fa niente per quattro cicli).

Ritardi più lunghi

La generazione di ritardi più lunghi tramite software può essere raggiunta attraverso l'uso di un contatore più ampio. Può essere usata una coppia di registri per contenere un conteggio a 16 bit. Per semplificare, presumiamo che il conteggio più basso sia "0". Il byte più basso sarà caricato con "255", il conteggio massimo, per poi passare attraverso un loop di decremento. Ogni qual volta è decrementato a "0", il byte superiore del contatore sarà decrementato di uno. Il programma termina ogni qual volta il byte superiore è decrementato al valore "0". Se è richiesta più precisione nella generazione del ritardo, il conteggio più basso può avere un valore non nullo. In questo caso, noi scriveremo il programma proprio come spiegato e aggiungeremo alla fine del programma le tre linee di generazione del ritardo, che è stato descritto sopra.

Qui sotto compare un programma di delay a 24 bit.

```
DEL 24    LD    B, COUNTH     CONTATORE ALTO (8 BIT)
DEL 16    LD    DE —1
LOOPA     LD    HL, COUNTL    CONTATORE BASSO
LOOPB     ADD   HL, DE        DECREMENTALO
          JR    C, LOOPB      VAI AVANTI FINO A NULLO
          DJNZ  LOOPA         DECREMENTA B E SALTA
```

Notate che DE è caricato con "—1", ed è usato per decrementare il contatore a 16 bit HL.

Naturalmente, usando più di tre parole potrebbero essere generati delay più lunghi. Ciò è analogo al funzionamento del contachilometri di un'automobile.

Quando la ruota più a destra va da "9" a "0", la prossima ruota a sinistra è incrementata di uno. Questo è il principio generale quando si conteggia con unità discrete multiple.

Comunque, lo svantaggio principale di questo metodo è che quando si conteggiano i ritardi, il microprocessore non farà nient'altro per centinaia di millisecondi o addirittura secondi. Se il computer non ha nient'altro da fare, questo è perfettamente accettabile.

Comunque, in generale il microcomputer dovrebbe essere disponibile per altri compiti, e quindi i ritardi più lunghi non saranno realizzati via software. Infatti, anche ritardi corti possono essere biasimevoli in un sistema se deve fornire un tempo di risposta garantito in date situazioni. Allora devono essere usati i ritardi hardware. Inoltre, se sono usati gli interrupt, può essere persa l'esattezza della temporizzazione se può essere interrotto il loop di conteggio.

Esercizio 6-3: *Scrivete un programma per realizzare un ritardo di 100 ms (tipico di una Telescrivente).*

Ritardi hardware

I ritardi hardware sono compiuti usando un *timer ad intervallo programmabile* o semplicemente "timer". Un registro del timer è caricato con un valore.

La differenza è che periodicamente, il timer decrementerà automaticamente il contatore. Il periodo può di solito essere regolato o scelto dal programmatore. Ogni qual volta il timer è decrementato a "0", invierà normalmente un interrupt al microprocessore. Può anche porre ad 1 un bit di stato che può essere percepito periodicamente dal computer. L'uso degli interrupt sarà spiegato più avanti in questo capitolo.

Altri modi di funzionamento del timer possono includere l'inizio da "0" e il conteggio della durata del segnale, oppure, il conteggio del numero di impulsi ricevuti. Quando funziona come un timer d'intervallo, si dice che il timer funziona in un modo ad *one-shot* (un colpo). Quando conta gli impulsi, si dice operi in un modo a *conteggio di impulsi*. Alcuni dispositivi timer possono includere anche registri multipli ed un numero di accessori opzionali che il programmatore può scegliere.

Rivelazione Di Impulsi

Il problema della rivelazione di impulsi è opposto alla generazione ed include un'altra difficoltà: mentre è generato un impulso di output sotto il controllo del programma, gli impulsi d'input avvengono in *modo asincrono* con il programma.

Per rivelare un impulso possono essere usati due metodi: il *polling* (registrazione) e gli *interrupt*. Gli interrupt saranno trattati più tardi in questo capitolo.

Adesso consideriamo la tecnica di polling. Usando questa tecnica, il programma legge continuamente il valore di un dato registro d'input, testando una posizione di bit, forse il bit 0. Sarà supposto che il bit 0 sia originariamente "0".

Ogni qual volta è ricevuto un impulso, questo bit assumerà il valore "1". Il programma controlla continuamente il bit 0 fino a quando assume il valore "1".

L'impulso è stato rivelato, quando è trovato un "1". Il programma è il seguente:

```
POLL    IN    A, (INPUT)    LEGGI IL REGISTRO D'INPUT
ON      BIT   0, A          TEST PER 0
        JR    Z, POLL       CONTINUA IL POLLING SE È 0
```

Viceversa, supponiamo che la linea d'input sia normalmente "1" e che si voglia rivelare uno "0". Questo è il caso normale per rivelare un bit START, quando si controlla una linea collegata ad una telescrivente. Il programma compare sotto:

```
POLL    IN    A, (INPUT)    LEGGI IL REGISTRO D'INPUT
        BIT   0,A           PONI AD 1 IL FLAG Z
        JR    NZ, POLL      IL TEST È INVERTITO
START
```

Monitoraggio della durata

Il controllo della durata di un impulso può essere eseguito nello stesso modo del calcolo della durata di un impulso di output. Può essere usata la tecnica hardware o software. Quando si controlla un impulso con la tecnica software, un contatore è regolarmente incrementato di 1 e poi è verificata la presenza dell'impulso. Il programma fa un loop su se stesso se l'impulso è ancora presente. Quando scompare l'impulso, il conteggio contenuto nel registro di contatore è usato per calcolare l'effettiva durata dell'impulso:

```
DURTN   LD    B, 0          AZZERA IL CONTATORE
AGAIN   IN    A, (INPUT)    LEGGI L'INPUT
        BIT   0, A          CONTROLLA IL BIT 0
        JR    Z, AGAIN      ASPETTA UN "1"
LONGER  INC   B             INCREMENTA IL CONTATORE
        IN    A, (INPUT)    CONTROLLA IL BIT 0
        BIT   0,A
        JR    NZ, LONGER    ASPETTA UNO "0"
```

Naturalmente, supponiamo che la durata massima dell'impulso non provochi l'overflow del registro B. Se questo fosse il caso, il programma dovrebbe essere cambiato per prenderlo in considerazione (oppure sarebbe un errore di programmazione!).

Poiché possiamo percepire e generare impulsi, catturiamo o trasferiamo quantità più grandi di dati.

Saranno distinti due casi: dati in serie o seriali e dati in parallelo o paralleli.

Successivamente applicheremo questa conoscenza a dispositivi d'input/output.

TRASFERIMENTO PARALLELO DELLE PAROLE

Supponiamo che otto bit di dati di trasferimento siano disponibili in parallelo all'indirizzo "INPUT" (vedere la Figura 6-4).

Il microprocessore deve leggere la parola dei dati in questa posizione ogni qual volta che una parola di stato indica che è valida. L'informazione di stato si supporrà sia contenuta nel bit 7 dell'indirizzo "STATUS". Qui scriveremo un programma che leggerà e salverà automaticamente ogni parola di dati quando entra. Per semplificare, noi supporremo che il numero delle parole da leggere sia noto in anticipo e contenuto nella posizione "COUNT".

Se questa informazione non fosse disponibile, verificheremo un *carattere di break* (interruzione), come un *rubout*, o forse il carattere "*". Questo l'abbiamo già imparato a fare.

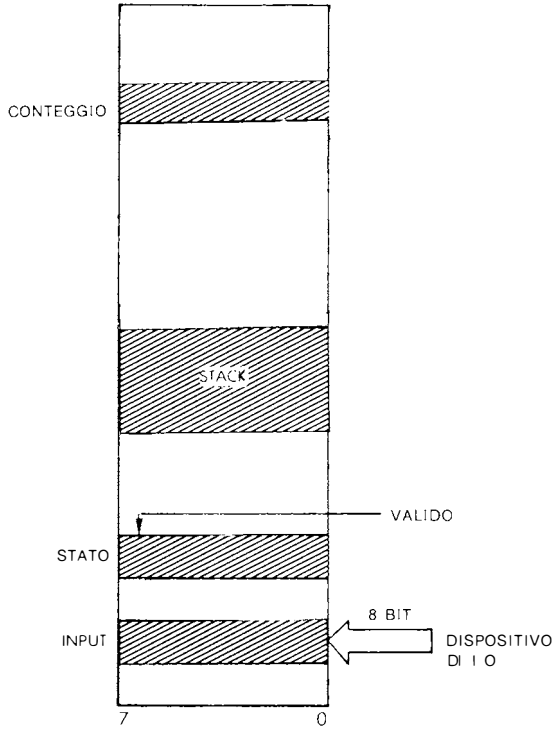


Figura 6-4: Trasferimento parallelo di parola: La Memoria.

Il diagramma di flusso compare nella Figura 6-5. È abbastanza facile. Noi proviamo l'informazione di stato fino a quando diventa "1", indicando che la parola è pronta. Quando la parola è pronta, la leggiamo e la conserviamo in una locazione di memoria appropriata. Quindi decrementiamo il contatore e proveremo se è diminuito a "0". Se è così, noi abbiamo finito: e se non è così, leggiamo la parola successiva.

Di seguito è riportato un semplice programma che esegue questo algoritmo:

```
PARAL   LD   A, (COUNT)   LEGGI IL CONTEGGIO IN A
        LD   B,A          B È IL CONTATORE
WATCH   IN   A, (STATUS)  CERCA "DATA READY" VERO
        BIT  7, A         BIT 7 È "1" SE IL DATO È PRONTO
        JP   Z, WATCH     DATO VALIDO?
        IN   A, (INPUT)   LEGGI IL DATO
        PUSH AF           CONSERVA IL DATO NELLO STACK
        DEC  B            DECREMENTA IL CONTEGGIO
        JP   NZ, WATCH    ESEGUI FINO A ZERO
```

È supposto che il flag di "data ready" è cancellato automaticamente quando è letto STATUS, come è di solito il caso su un controllore di dispositivi. Le prime due istruzioni inizializzano il registro contatore B:

```
PARAL   LD A, (COUNT)
        LD   B, A
```

Notate che non c'è nessun mezzo facile per caricare solo B dalla memoria. Si deve caricare A, poi trasferire il suo contenuto a B, oppure caricare B e C simultaneamente.

POLLING O RICHIESTA DI SERVIZIO

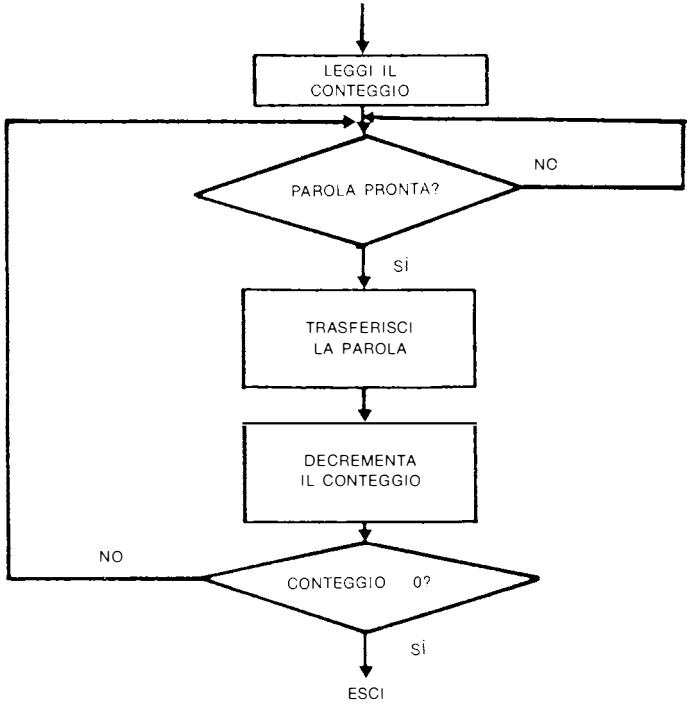


Figura 6-5: Trasferimento parallelo di parole: Il Diagramma di Flusso

Le successive tre istruzioni del programma leggono le informazioni di stato e provocano un loop fino a quando il bit sette del registro di stato è "0". (È il bit di segno, cioé, bit N).

```
IN      A, (STATUS)
BIT     7,A              "IN" NON PONE AD UNO I FLAG
JP      Z, WATCH
```

Quando il JP non riesce, il dato è valido e possiamo leggerlo:

```
IN      A, (INPUT)
```

La parola ora è stata letta dall'indirizzo INPUT dove era, e deve essere conservata. Supponendo sia disponibile un'area di stack sufficiente, noi possiamo usare:

```
PUSH    AF
```

che salva A (ed F) nello stack. Se lo stack è pieno, o il numero di parole che deve essere trasferito è grande, non possiamo caricarle sullo stack e dobbiamo perciò trasferirle ad una area di memoria designata usando, per esempio, una istruzione indicizzata. Comunque, questo ri-

chiederebbe una istruzione extra per incrementare o decrementare il registro indice. PUSH è più veloce (solo 11 cicli di clock).

La parola dati è stata adesso letta e conservata.

Decrementeremo semplicemente il contatore delle parole e proveremo se abbiamo terminato:

```
DEC     B
JP      NZ,WATCH
```

Noi manterremo la situazione di loop fino a quando alla fine il contatore decrementa a "0".

Questo programma da nove istruzioni può essere chiamato *benchmark* (segno di riferimento). Il programma benchmark è un programma attentamente ottimizzato, progettato per provare le capacità di un dato processore in una situazione specifica.

I trasferimenti paralleli sono una di queste situazioni tipiche.

Questo programma è stato progettato per la velocità e efficienza massima. Adesso calcoliamo la velocità massima di trasferimento di questo programma. Supporremo che COUNT sia contenuto nella memoria. La durata di ogni istruzione è determinata esaminando la tabella alla fine del libro ed è trovata essere la seguente:

```
PARAL   LD      A,(COUNT)   13
        LD      B,A         4
WATCH   IN      A,(STATUS)  11
        BIT     7,A         8
        JP      Z,WATCH     7/12
        IN      A,(INPUT)   11
        PUSH    AF          11
        DEC     B           4
        JP      NZ,WATCH    7/12
```

Il tempo minimo d'esecuzione è ottenuto supponendo che il dato è disponibile ogni volta che componiamo STATUS. In altre parole, il primo JP si suppone non riesca ogni volta. Dunque la temporizzazione è:

$$13 + 4 + (11 + 8 + 7 + 11 + 11 + 4 + 7) \times \text{COUNT}$$

Tralasciando i primi 17 cicli necessari ad inizializzare il registro del contatore, il tempo usato per trasferire una parola è 59 cicli di clock oppure 29,5 microsecondi con un clock da 2 MHz. Perciò, la velocità massima di trasferimento dei dati è:

$$\frac{1}{29{,}5\ (10^6)} = 33 \text{ Kbyte per secondo}$$

Esercizio 6-4: *Supponiamo che il numero di parole che deve essere trasferito sia maggiore di 256. Modificate perciò il programma e determinate l'impatto sulla velocità massima di trasferimento dei dati.*

Esercizio 6-5: *Modificate questo programma per cercare di migliorare la sua velocità:*

1 - usando JR invece di JP
2 - usando DJNZ
3 - usando INIR o INDR

Il programma precedente era veramente ottimale?

Adesso abbiamo imparato ad eseguire trasferimenti paralleli ad alta velocità. Consideriamo un caso più complesso.

TRASFERIMENTO SERIALE DI BIT

Un input seriale è quello in cui i bit dell'informazione (0 oppure 1) entrano successivamente su una linea.

Questi bit possono entrare ad intervalli regolari. Questa è normalmente chiamata trasmissione *sincrona*. Oppure, possono arrivare ad intervalli casuali. Questa è chiamata trasmissione *asincrona*. Svilupperemo un programma che possa funzionare in entrambi i casi. Il principio della cattura dei dati sequenziali è semplice: guarderemo una linea d'input, che noi supporremo sia la linea 0. Quando è rilevato un bit di dati su questa linea, leggeremo il bit e lo sposteremo in un registro temporaneo. Ogni qual volta sono assemblati otto bit, noi conserveremo il byte dei dati nella memoria e assembleremo il prossimo byte. Per semplificare, supporremo che il numero di byte che deve essere ricevuto sia conosciuto in anticipo. Altrimenti, per esempio, potremmo dover guardare a un carattere di break speciale ed interrompere a questo punto il trasferimento in serie dei bit. Abbiamo imparato a farlo. Il diagramma di flusso per questo programma è riportato nella Figura 6-6.

Il programma è il seguente:

```
SERIAL   LD     C,0          CANCELLA LA PAROLA D'INPUT
         LD     A,(COUNT)    CARICA B CON IL CONTEGGIO DEI BYTE
         LD     B,A
LOOP     IN     A,(INPUT)    LEGGI LA SORGENTE
         BIT    7,A          IL BIT 7 È LO STATO, IL BIT 0 È IL DATO
         JR     Z,LOOP       ASPETTA UN "1"
         SRL    A            SPOSTA IL BIT DEI DATI NEL RIPORTO
         RL     C            SALVA L'INPUT B IN C
         JR     NC,LOOP      CONTINUA FINO A QUANDO
                             SONO DENTRO 8 BIT
         PUSH   BC           CONSERVA LA PAROLA NELLO STACK
         LD     C,01H        RESET DEL MARKER BIT
         DEC    B            DECREMENTA IL CONTATORE DI BYTE
         JR     NZ,LOOP      ASSEMBLA LA PROSSIMA PAROLA
```

Questo programma è stato progettato con caratteristiche di efficienza e userà nuove tecniche che spiegheremo (vedere la Figura 6-7).

Le convenzioni sono le seguenti: la locazione di memoria COUNT è supposta contenere un conteggio del numero di parole che deve essere trasferito. Il registro C sarà usato per assemblare otto bit consecutivi che entrano. L'indirizzo INPUT si riferisce ad un registro d'input. È supposto che la posizione di bit 7 di questo registro sia un flag di stato, o un bit di clock. Quando è "0", il dato non è valido.

Il dato è valido quando è "1". Il dato stesso si suppone apparirà nella posizione di bit 0 di questo stesso registro. In molti casi, l'informazione di stato apparirà su un registro differente dal registro dei dati. Di conseguenza, modificare questo programma dovrebbe essere allora un compito semplice. Inoltre, supporremo che il primo bit dei dati ad essere ricevuto da questo programma sia un "1". Esso indica che segue il dato reale. Se questo non fosse, vedremo in seguito un'ovvia modificazione per tener conto di ciò. Il programma corrisponde esattamente al diagramma di flusso della Figura 6-6. Le prime linee del programma compiono un loop d'attesa che prova se è pronto un bit. Per determinare se è pronto un bit, leggiamo il registro d'input, poi proviamo il bit zero (Z). L'istruzione JR riuscirà fino a quando questo bit è "0" e torneremo indietro al loop. Ogni qual volta il bit di stato (o ciclo di clock) diventa vero ("1"), allora JR non riuscirà e sarà eseguita l'istruzione successiva.

Questa sequenza iniziale di istruzioni corrisponde alla freccia 1 nella Figura 6-7.

A questo punto, l'accumulatore contiene un "1" nella posizione bit 7 e l'attuale bit di dati nella posizione dei bit 0. Il primo bit di dati ad arrivare sarà "1". Comunque, i bit successivi possono essere "0" oppure "1". Ora vogliamo conservare il bit di dati che è stato raccolto nella posizione 0.

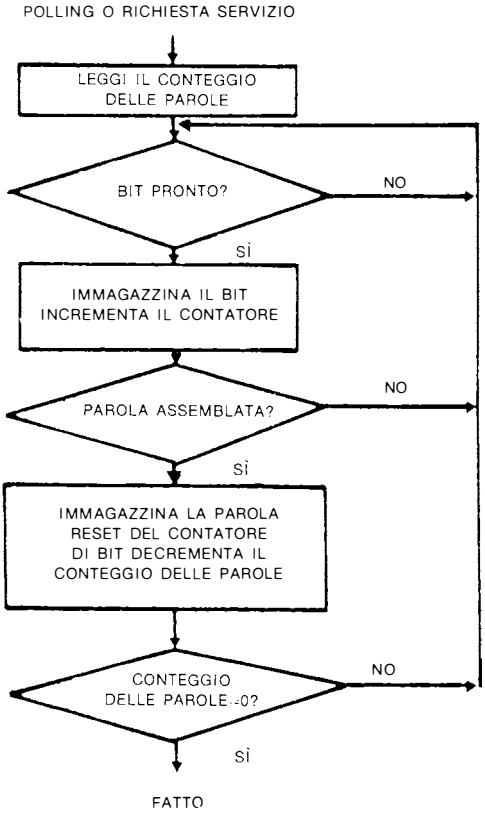


Figura 6-6: Trasferimento in serie di bit — Il diagramma di flusso

L'istruzione:

SRL A

sposta il contenuto dell'accumulatore a destra di una posizione. Questa fa si che il bit più a destra di A, che è il bit di dati, cada nel bit di riporto. Ora conserveremo questo bit di dati nel registro C (questo processo è illustrato dalle frecce 2 e 3 nella Figura 6-7)

RL C

L'effetto di questa istruzione è quello di leggere il bit di riporto nella posizione di bit più a destra di C. Allo stesso tempo, il bit più a sinistra di C cade nel bit di riporto. (Riferitevi al Capitolo 4 se avete qualche dubbio sulla operazione di rotazione).

È importante ricordare che una operazione di rotazione salverà il bit di riporto, nella posizione bit più a destra e condizionerà anche il bit di riporto con il valore del bit 7.

Qui, uno "0" cadrà nel riporto. La prossima istruzione:

JR NC, LOOP

prova il riporto e si ramifica indietro all'indirizzo LOOP fino a quando il riporto è "0". Questo è il nostro contatore automatico di bit. Si può osservare che, come risultato del primo RL, C con-

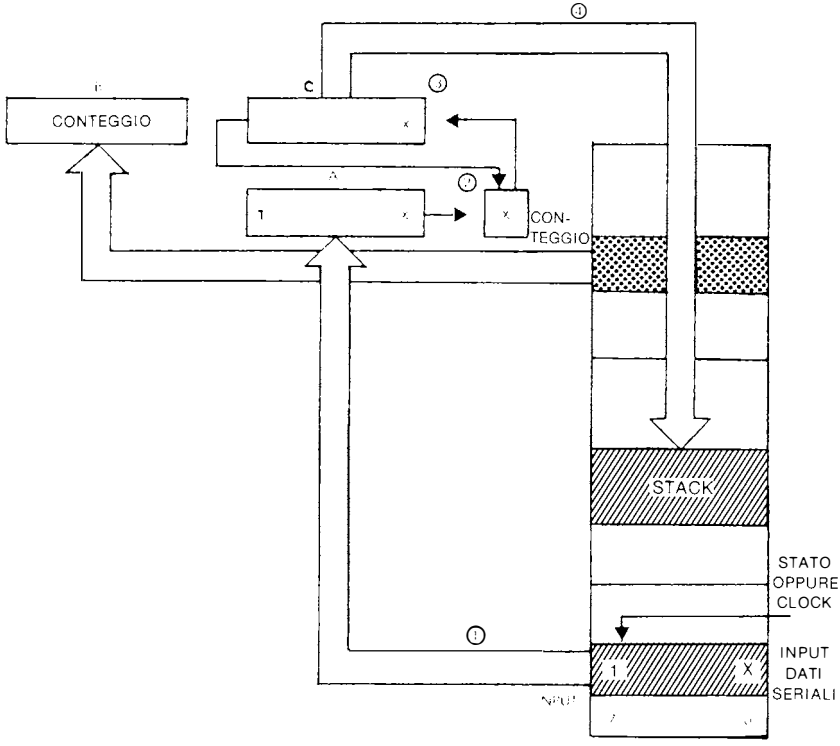


Figura 6-7: Trasformazione serie-parallelo — I registri

terrà "00000001". Otto spostamenti dopo, l'"1" cadrà finalmente nel bit di riporto ed arresterà la ramificazione. Questa è una maniera ingegnosa per realizzare un contatore di loop automatico senza dover consumare una istruzione per decrementare il contenuto di un registro indice. Questa tecnica è usata per abbreviare il programma e migliorare le sue prestazioni. Quando alla fine JR NC non riesce, 8 bit saranno stati assemblati in C. Questo valore dovrebbe essere conservato nella memoria. Questo è compiuto dall'istruzione successiva (freccia 4 sulla Figura 6-7):

PUSH BC

Qui stiamo salvando il contenuto di B e C nello stack. Il salvataggio nello stack è possibile solo se c'è spazio sufficiente. A patto che questa condizione sia soddisfatta, questo è di solito il modo più veloce per conservare una parola nella memoria anche se, in questo caso, salviamo un registro non necessario (B). Il puntatore dello stack è aggiornato automaticamente. Se non stessimo spingendo una parola nello stack, dovremmo usare un'altra istruzione per aggiornare il puntatore di memoria. Noi potremmo equivalentemente eseguire una operazione con indirizzamento indicizzato, ma ciò implicherebbe anche il decremento e l'incremento dell'indice, usando tempo extra.

Dopo che è stata salvata la prima parola di dati, non ci sarà più nessuna garanzia che il primo bit di dati ad entrare sia un "1". Potrebbe essere qualsiasi numero.

Perciò, dobbiamo eseguire il reset del contenuto a "00000001" in modo che possiamo continuare ad usarlo come un contatore di bit. Questo è eseguito dalla prossima istruzione:

```
LD   C,01H
```

Alla fine, decrementeremo il contatore di parola, poichè è stata assemblata una parola e proveremo se abbiamo raggiunto la fine del trasferimento. Questo è compiuto dalle prossime due istruzioni:

```
DEC  B
JR   NZ,LOOP
```

Il programma precedente è stato progettato con caratteristiche di velocità, in modo che si possa catturare un veloce flusso di bit dei dati in ingresso. Una volta che termina il programma, è consigliabile prelevare dallo stack le parole che sono state conservate e trasferirle da qualche altra parte entro la memoria. Nel Capitolo 2 abbiamo già imparato ad eseguire un tale trasferimento di blocchi.

Esercizio 6-6: *Calcolate la velocità massima alla quale questo programma sarà capace di leggere i bit in serie. Guardate il numero di cicli richiesti da ogni istruzione nella tabella alla fine di questo libro, poi calcolate il tempo che trascorrerà durante l'esecuzione di questo programma. Per calcolare il tempo impiegato da un loop, moltiplicate semplicemente la durata totale di questo loop, espressa in microsecondi, per il numero di volte che sarà eseguito. Quando calcolate la velocità massima, supponete anche che un bit di dati sia pronto ogni volta che viene rivelata la locazione d'input.*

Questo programma è più difficile da capire dei precedenti.

Guardiamolo di nuovo (riferitevi alla Figura 6-6) più dettagliatamente, esaminando alcuni compromessi.

Di tanto in tanto un bit di dati entra nella posizione di bit 0 d'"INPUT". Per esempio, ci potrebbero essere tre "1" in successione.

Perciò, dobbiamo *differenziare tra i bit successivi* che entrano.

Questa è la funzione del segnale di clock. Il segnale di clock (o di STATUS) ci dice che il bit d'input adesso è valido. Perciò proveremo il bit di stato prima di leggere un bit. Se lo stato è "0", dobbiamo aspettare. Se è "1", allora il bit di dati è valido. Qui supponiamo che il segnale di stato sia collegato al bit 7 del registro INPUT.

Esercizio 6-7: *Sapete spiegare perchè il bit 7 è usato per lo stato, ed il bit 0 per i dati, È importante?*

Una volta che abbiamo catturato un bit di dati, vogliamo conservarlo in una posizione sicura, poi spostarlo a sinistra, in modo che possiamo prendere il bit successivo.

Sfortunatamente, l'accumulatore è usato per leggere e provare sia i dati che lo stato in questo programma. Se dovessimo accumulare dati nell'accumulatore, la posizione di bit 7 sarebbe cancellata dal bit di stato.

Esercizio 6-8: *Sapete suggerire un modo per provare lo stato senza cancellare il contenuto dell'accumulatore (una istruzione speciale?) Se può essere fatto, potremmo usare l'accumulatore per accumulare i bit successivi che entrano? Potete migliorare la velocità usando un "salto automatizzato"?*

Esercizio 6-9: *Riscrivete il programma, usando l'accumulatore per immagazzinare i bit che entrano. Confrontatelo con il precedente in termini di velocità e numero d'istruzioni.*

Indirizziamo altre due possibili variazioni.

Nel nostro particolare esempio, abbiamo supposto che il primo bit ad entare sia un segnale speciale, garantito essere "1". Comunque, in generale, può essere qualunque cosa.

Esercizio 6-10: *Modificate il programma precedente, supponendo che il primo bit ad entrare sia un dato valido (da non essere scartato), e può essere "0" oppure "1". Avviso: il nostro "contatore di bit" dovrebbe ancora funzionare correttamente, se lo inizializzate con il valore corretto.*

Alla fine, per guadagnare tempo abbiamo la parola assemblata nello stack. Naturalmente, potremmo salvarla in un'area di memoria specifica.

Esercizio 6-11: *Modificate il programma precedente e salvate la parola assemblata nell'area di memoria iniziando alla BASE.*

Esercizio 6-12: *Modificate il programma precedente in modo che il trasferimento si interrompa quando nel flusso di input è rivelato il carattere "S".*

L'Alternativa Hardware

Analogamente alla maggior parte degli algoritmi di input/output standard, è possibile compiere questa procedura tramite hardware. Il chip è chiamato UART ed accumulerà automaticamente i bit. Comunque, quando si desidera ridurre il numero di componenti, sarà invece usato questo programma, o una sua variante.

Esercizio 6-13: *Modificate il programma, supponendo che il dato sia disponibile nella posizione bit 0 della locazione INPUT, mentre l'informazione di stato è disponibile nella posizione bit 0 dell'indirizzo INPUT + 1.*

SOMMARIO FONDAMENTI DI I/O

Abbiamo imparato ad eseguire operazioni d'input/output elementari ed a trattare un flusso di dati paralleli o bit in serie. Adesso siamo pronti a comunicare con dispositivi di input/output reali.

COMUNICAZIONE CON I DISPOSITIVI DI INPUT/OUTPUT

Per scambiare dati con i dispositivi d'input/output, per prima cosa dobbiamo assicurarci che il dato sia disponibile, se lo vogliamo leggere; oppure se il dispositivo è pronto ad accettare il dato, se vogliamo inviarlo. Si possono usare due procedure: handshaking (stretta di mano) ed interrupt.

Handshaking

L'handshaking (Stretta di mano) è generalmente usato per la comunicazione tra due dispositivi asincroni cioè tra due dispositivi che non sono sincronizzati. Per esempio, se vogliamo inviare una parola ad una stampante parallela, dobbiamo prima essere sicuri che sia disponibile il buffer d'input (memoria di transito) di questa stampante.

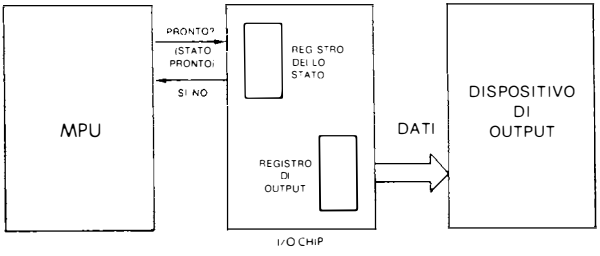


Figura 6-8: Handshaking (Output)

Così, chiederemo alla stampante: Sei pronta? La stampante dirà "sì" o "no". Se non è pronta aspetteremo. Se è pronta, invieremo i dati (vedere la Figura 6-8).

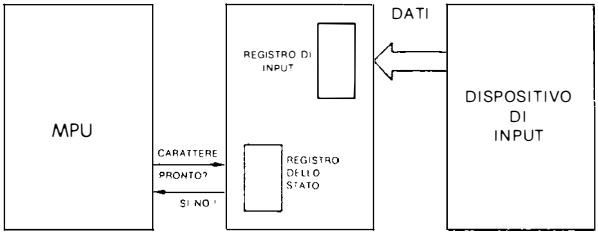


Figura 6-8a: Handshaking (Input)

Viceversa, verificheremo se il dato è valido prima di leggere il dato da un dispositivo d'input. Chiederemo: È valido il dato? Ed il dispositivo ci dirà "sì" o "no". Il "sì o no" può essere indicato dai bit di stato o da altri mezzi (vedere la Figura 6-8a). Come analogia dovreste essere sicuri che sia pronto a comunicare con voi, ogni qual volta desiderate scambiare informazioni con qualcuno che è indipendente o che al momento sta facendo qualcos'altro.

La regola solita di cortesia è quella di stringergli la mano.

Successivamente può seguire lo scambio di dati. Questa è la procedura normalmente usata per comunicare con i dispositivi d'input/output.

Illustriamo questa procedura con un semplice esempio.

Invio di un Carattere alla Stampante

Si suppone che il carattere sia contenuto nella posizione di memoria CHAR. Il programma per la stampa è il seguente:

```
WAIT    IN      A, (STATUS)
        BIT     7, A          PROVA SE È PRONTO
        JR      Z, WAIT       ALTRIMENTI ASPETTA
        LD      A, (CHAR)     PRENDI IL CARATTERE
        OUT     (PRNTD), A    STAMPALO
        JR      WAIT          CONSIDERA IL PROSSIMO
```

Il programma di stampa è facile e usa la procedura handshaking descritta. I percorsi dei dati sono mostrati nella Figura 6-9.

Il carattere (chiamato DATA) è sistemato alla locazione di memoria CHAR. Per prima cosa, viene controllato lo stato della stampante. Ogni volta che il bit 7 del registro di stato diventa 1, significa che la stampante è pronta per l'output, cioè, è disponibile il suo buffer di output. A questo punto, il carattere è caricato nell'accumulatore, poi fatto uscire sulla stampante, tramite l'accumulatore.

Fino a quando il bit di stato rimane 0, il programma rimarrà in un loop, chiamato WAIT nel programma.

Esercizio 6-14: *Quante istruzioni si risparmierebbero nel programma precedente se fosse possibile caricare i dati direttamente nel registro C oppure fare uscire il contenuto del registro C direttamente?*

Esercizio 6-15: *Quando usate una stampante reale, di solito è necessario inviare un ordine di start (inizio) prima di usare il dispositivo. Modificate questo programma per generare un tale ordine, supponendo che il comando di start sia ottenuto scrivendo un 1 nella posizione bit 0 del registro di STATUS, il quale è supposto essere bidirezionale.*

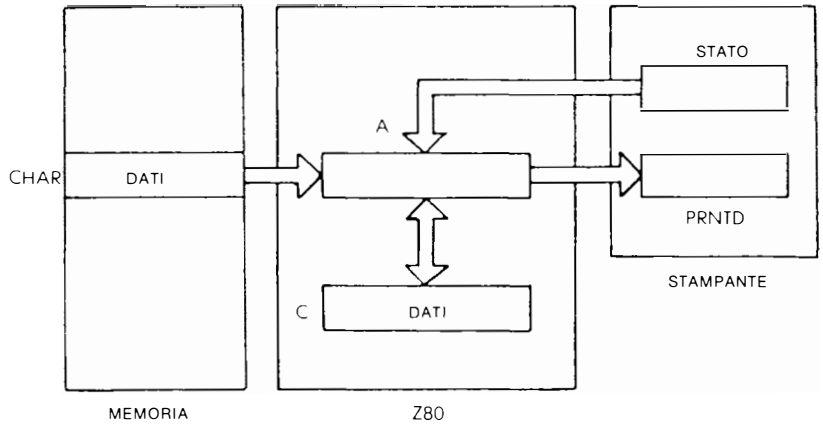


Figura 6-9: Stampante — I percorsi dei dati

ESERCIZIO 6-16: *Se l'istruzione BIT non fosse disponibile, potreste usare invece un'altra istruzione, nella linea 4 del programma? Se è possibile, spiegate il vantaggio dell'uso dell'istruzione BIT.*

Esercizio 6-17: *Modificate il programma precedente per stampare una stringa di caratteri n, dove si suppone che n sia minore di 255.*

Esercizio 6-18: *Modificate il programma precedente per stampare una stringa di caratteri fino a quando viene incontrato un codice di ritorno carrello.*

Adesso complichiamo la procedura di output richiedendo una conversione di codice ed eseguendo l'output contemporaneamente su dispositivi diversi.

Output su un LED a sette Segmenti

Un diodo emettitore di luce (LED) a sette segmenti può mostrare le cifre da "0" a "9", oppure i digit esadecimali da "0" ad "F" tramite combinazioni d'illuminazione dei suoi sette segmenti. Nella Figura 6-10 è mostrato un LED a sette segmenti. Nella Figura 6-11 compaiono i caratteri che possono essere generati con questo LED.

Nella Figura 6-10 i segmenti di un LED sono contrassegnati con le lettere da "a" a "g". Per esempio, "0" sarà rappresentato illuminando i segmenti a b c d e f. Supponiamo adesso, che il bit "0" di una porta di output sia collegato al segmento "a", e che "1" sia collegato al segmento "b", e così via. Il bit 7 non è usato. Il codice binario richiesto per illuminare f e d c b a (per mostrare "0") è perciò: "0111111". Nell'esadecimale questo è "3F". Fate l'esercizio seguente.

Esercizio 6-19: *Calcolate l'equivalente a sette segmenti dei digit esadecimali da "0" a "F". Riempite la tabella seguente:*

Esadec.	Codice LED	Esadec.	Codice LED	Esadec.	Codice LED	Esadec.	Codice LED
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Ora mostriamo i valori esadecimali su *diversi* LED.

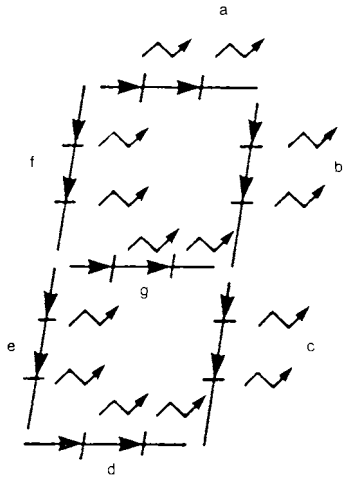


Figura 6-10: LED a sette segmenti

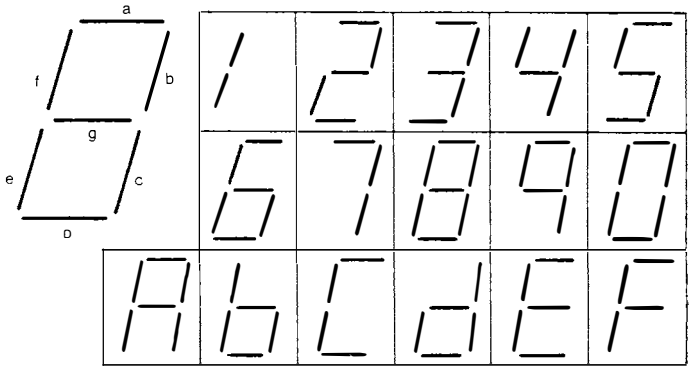


Figura 6-11: Caratteri esadecimali generati con un LED a sette segmenti

Pilotaggio di LED Multipli

Un LED non ha memoria. Mostrerà i dati solo fino a quando sono attive le sue linee dei segmenti. Per mantenere basso il costo di un display LED, il microprocessore mostrerà le informazioni alternativamente *su ognuno dei LED.*

La rotazione tra i LED deve essere abbastanza veloce in modo che non ci sia nessun lampeggiamento apparente. Questo implica che il tempo impiegato da un LED sia minore di 100 millisecondi. Progettiamo un programma che lo compia. Il registro C sarà usato per puntare al LED sul quale vogliamo mostrare un digit. Si suppone che l'accumulatore contenga il valore esadecimale che deve essere mostrato sul LED. La nostra prima premura è quella di convertire il valore esadecimale nella sua rappresentazione a sette segmenti. Nel capitolo precedente,

abbiamo costruito la tabella delle equivalenze. Poichè stiamo accedendo ad una tabella, useremo il modo di indirizzamento indicizzato, dove l'indice di spostamento sarà fornito dal valore esadecimale.

Ciò significa che il codice a sette segmenti per il digit esadecimale "3" viene ottenuto consultando il terzo elemento della tabella dopo la base. L'indirizzo della base sarà chiamato SEGBAS. Il programma è il seguente:

```
LEDS    LD      E, A            A CONTIENE IL DIGIT ESADECIMALE
        LD      D, 0            USA "DE" COME SPOSTAMENTO
        LD      HL, SEGBAS      USA "HL" COME INDICE
        ADD     HL, DE          INDIRIZZO DELLA TABELLA
        LD      A, (HL)         LEGGI CODICE DELLA TABELLA
        LD      B, 50H          VALORE DI RITARDO = QUALSIASI
                                NUMERO ELEVATO
        OUT     (C), A          OUTPUT PER LA DURATA IMPOSTATA
DELAY   DEC     B               CONTATORE DEL RITARDO
        JR      NZ, DELAY       CONTINUA IL LOOP
        DEC     C               C È IL NUMERO DELLA PORTA
        LD      A, C
        CP      MINLED          FATTO PER L'ULTIMO LED?
        JR      NZ, OUT
        LD      BC, (MAXLED)    SE È COSÌ, RESET DI C AL PRIMO LED
OUT     RET
```

Il programma suppone che il registro C contenga l'indirizzo del successivo LED che deve essere illuminato e che l'accumulatore A contenga il digit che deve essere mostrato.

Il programma per prima cosa consulta il codice a sette segmenti che corrisponde al valore esadecimale contenuto nell'accumulatore. I registri D ed E vengono usati come un campo di spostamento, e i registri H e L sono usati come registro indice a 16 bit. Il codice per il digit esadecimale è sommato all'indirizzo di base della tabella:

```
LEDS    LD      E, A            CODICE A 7 SEGMENTI
        LD      D, 0
        LD      HL, SEGBAS
        ADD     HL, DE
```

Poi viene eseguito un loop di ritardo, in modo che il codice ottenuto dalla tabella è mostrato per una durata appropriata. Qui la costante esadecimale "50" è stata scelta arbitrariamente:

```
        LD      A, (HL)         LEGGERE IL CODICE DALLA TABELLA
        LD      B, 50H          VALORE DEL RITARDO
```

Il ritardo è compiuto usando un loop di ritardo classico. La prima istruzione:

```
        OUT     (C), A
```

fa uscire il contenuto dell'accumulatore alla porta di I/O puntata dal registro C (il numero del LED). Le prossime due istruzioni compiono il loop di ritardo:

```
DELAY   DEC     B
        JR      NZ, DELAY
```

Una volta che il ritardo è stato compiuto, dobbiamo decrementare semplicemente il puntatore

al LED, e assicurarci di tornare all'indirizzo LED più alto se è stato raggiunto l'indirizzo LED più basso:

```
        DEC     C
        LD      A,C
        CP      MINLED
        JR      NZ,OUT
        LD      BC,(MAXLED)
OUT     RET
```

Qui si suppone che il programma sia stato scritto come subroutine ed allora l'ultima istruzione è RET: "ritorno da subroutine".

Esercizio 6-20: *Di solito è necessario spegnere i driver dei segmenti LED prima di mostrare il digit. Modificate il programma precedente aggiungendo le istruzioni necessarie (fare uscire "00" come codice di carattere prima di fare uscire il carattere).*

Esercizio 6-21: *Cosa accadrebbe al display se la label DELAY fosse fatta salire di una posizione? Questo, cambierebbe la temporizzazione? Questo varierebbe l'aspetto del display?*

Esercizio 6-22: *Noterete che le prime quattro istruzioni del programma stanno compiendo un accesso in memoria indicizzato a 16 bit. Infatti il meccanismo sembra rozzo, senza usare il meccanismo di indicizzazione. Supponete che l'indirizzo SEGBAS sia noto in anticipo. Chiamate SEGBSH la parte alta di questo indirizzo, e SEGBSL la parte bassa di questo indirizzo. Immagazzinate SEGBSH nella parte alta del registro IX. Ora scrivete il programma precedente usando il meccanismo d'indirizzamento ad indice per lo Z80, usando SEGBSL come il campo di spostamento della istruzione. Quali sono i vantaggi e gli svantaggi di questo approccio?*

Esercizio 6-23: *Supponendo che il programma precedente sia una subroutine, noterete che usa i registri B, D, E, H ed L al suo interno e modifica il loro contenuto. Se la subroutine può liberamente usare l'area di memoria designata dall'indirizzo T1, T2, T3, T4, T5, potreste aggiungere istruzioni all'inizio e alla fine di questo programma per garantire che, quando la subroutine ritorna, il contenuto dei registri B, D, E, H ed L sia lo stesso di quando la subroutine era stata chiamata?*

Esercizio 6-24: *Lo stesso esercizio come sopra, ma supponete che l'area di memoria T1, ecc., non sia disponibile alla subroutine (Consiglio: ricordatevi che in ogni computer c'è un meccanismo incorporato per preservare le informazioni in un ordine cronologico).*

Adesso abbiamo risolto i problemi comuni d'input/output.
Consideriamo il caso di un periferico comune: la Telescrivente.

Input/Output su Telescrivente

La Telescrivente è un dispositivo seriale. Invia e riceve parole d'informazioni in un formato seriale. Ogni carattere è codificato in un formato ASCII ad 8 bit (la tabella ASCII compare alla fine di questo libro). Inoltre, ogni carattere è preceduto da uno "start" bit, e terminato con due "stop" bit. Nel caso dell'interfaccia 20 milliamp currente loop, che è usata frequentemente, lo stato della linea è normalmente "1". Questo è usato per indicare al processore che la linea non è stata tagliata. Uno "start" è una transizione da "1" a "0". Indica al dispositivo ricevente che seguono bit di dati. La Telescrivente standard è un dispositivo a 10 caratteri al secondo. Abbiamo appena stabilito che ogni carattere richiede 11 bit. Questo significa che la Telescrivente trasmetterà 110 bit al secondo. Si dice sia un dispositivo a 110 baud.

Progetteremo un programma per serializzare i bit da inviare alla Telescrivente alla velocità corretta.

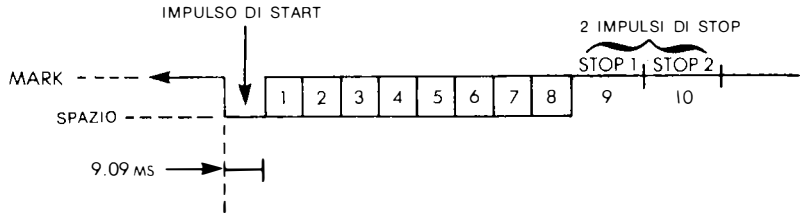


Figura 6-12: Formato di una parola della Telescrivente

110 bit al secondo implica che i bit sono separati da 9.09 millisecondi. Questa dovrà essere la durata del loop di ritardo che deve essere compiuto tra i bit successivi. Il formato di una parola di Telescrivente è riportato nella Figura 6-12. Nella Figura 6-13 compare il diagramma di flusso per l'input di bit. Il programma è il seguente:

```
TTYIN   IN      A, (STATUS)
        BIT     7, A          DATI PRONTI?
        JR      Z, TTYIN      ALTRIMENTI ASPETTA
        CALL    DELAY 1       CENTRO DELL'IMPULSO
        IN      A, (TTYBIT)   START BIT
        OUT     (TTYBIT), A   INVIA L'ECO
        CALL    DELAY 9       PROSSIMO IMPULSO (9 MS)
        LD      B, 08H        CONTEGGIO DI BIT
NEXT    IN      A, (TTYBIT)   LEGGI BIT DI DATI
        OUT     (TTYBIT), A   INVIA L'ECO
        SRL     A             SALVALO NEL RIPORTO
        RR      C             CONSERVALO IN C
        CALL    DELAY 9       PROSSIMO IMPULSO (9MS)
        DEC     B             DECREMENTA IL CONTEGGIO DI BIT
        JR      NZ, NEXT
        IN      A, (TTYBIT)   LEGGI IL BIT DI STOP
        OUT     (TTYBIT), A   INVIA L'ECO
        CALL    DELAY 9       SALTA IL SECONDO STOP
        RET
```

Figura 6-14: Programma di Telescrivente

Esaminiamo il programma dettagliatamente. Innanzitutto deve essere provato lo stato della Telescrivente per determinare se è disponibile un carattere:

```
TTYIN   IN      A, (STATUS)
        BIT     7, A
        JR      Z, TTYIN
```

L'istruzione "BIT" dello Z80 è molto utile in quanto permette di verificare qualsiasi bit in ogni registro di dati. Non modifica il contenuto del registro sotto test. Il flag Z è posto ad 1 se il bit specificato è 0, altrimenti è posto a 0.

Perciò, questo programma ciclerà fino a quando alla fine il bit di stato diventa "1". È un classico loop di polling.

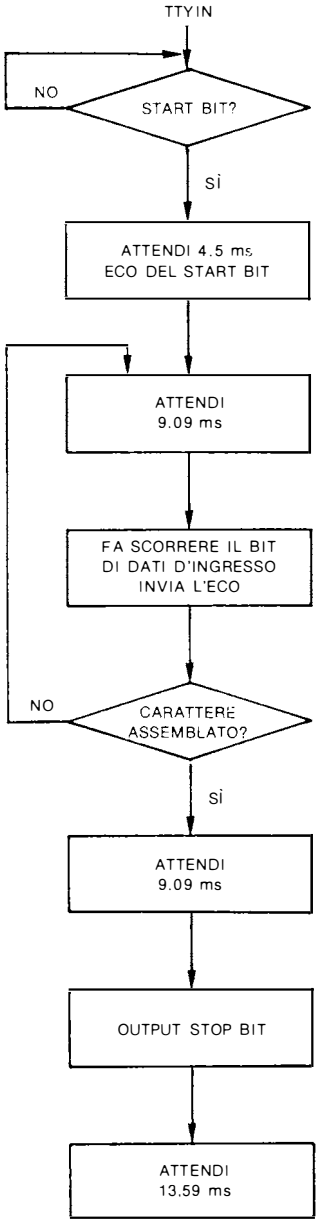


Figura 6-13: Input dalla telescrivente con Eco

Poiché lo STATUS non ha bisogno di essere conservato, potremmo usare vantaggiosamente:

```
        AND     10000000B
```

invece di

```
        BIT     7, A
```

L'uso dell'istruzione AND fa risparmiare due byte. Comunque, distrugge il contenuto di A (che qui è accettabile).

Quando si ottimizza un programma, ricordatevi che ogni nuova istruzione può introdurre effetti collaterali.

Quindi viene realizzato un ritardo di 4.5 ms per rivelare lo start bit al centro dell'impulso.

```
        CALL    DELAY 1
```

dove DELAY 1 è la subroutine di ritardo che realizza il ritardo richiesto. Il primo bit ad arrivare è lo start bit. Occorre inviare l'eco alla Telescrivente, altrimenti è ignorato. Questo è fatto dalle istruzioni successive:

```
TTYIN   IN      A (TTYBIT)
        OUT     (TTYBIT), A
```

Poi dobbiamo aspettare il primo bit di dati. Il ritardo necessario è uguale a 9.09 millisecondi ed è compiuto da una subroutine:

```
        CALL    DELAY9
```

Il registro B è usato come contatore ed è caricato con il valore 8 per catturare gli 8 bit di dati:

```
        LD      B, 08H
```

Quindi ogni bit di dati sarà letto nell'accumulatore, e poi sarà inviato l'eco. Si suppone di arrivare nella posizione di bit 0 dell'accumulatore. Il bit di dati sarà poi conservato nel registro C, dove sarà spostato.

Il trasferimento da A a C è compiuto attraverso il bit di riporto:

```
NEXT    IN      A, (TTYBIT)
        OUT     (TTYBIT), A
        SRL     A
        C       C
```

Questa sequenza è illustrata nella Figura 6-15.

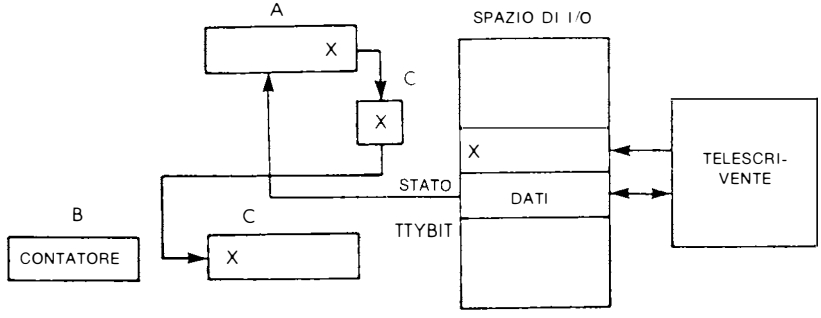


Figura 6-15: Input da Telescrivente

Quindi viene eseguito il solito ritardo di 9 millesecondi, il contatore di bit è decrementato ed il loop è eseguito di nuovo fino a quando non sono stati catturati gli otto bit:

```
CALL    DELAY 9
DEC     B
JR      NZ, NEXT
```

Alla fine, è catturato il bit STOP e viene inviato l'eco. È di solito sufficiente inviare un bit STOP singolo, comunque potrebbero essere entrambi rimandati indietro usando altre due istruzioni:

```
IN      A, (TTYBIT)
OUT     (TTYBIT), A)
CALL    DELAY9
RET
```

Il programma dovrebbe essere esaminato con attenzione. La logica è piuttosto semplice. Il fatto nuovo è che ogni volta che un bit è letto dalla Telescrivente (all'indirizzo TTYBIT), viene inviato l'eco alla Telescrivente stessa. Questa è una caratteristica standard della Telescrivente. Ogni volta che chi la usa preme un tasto, l'informazione è trasmessa al processore e poi indietro al meccanismo di stampa della Telescrivente. Questo dimostra che le linee di trasmissione stanno funzionando e che il processore sta operando quando si ha la stampa di un carattere in modo corretto.

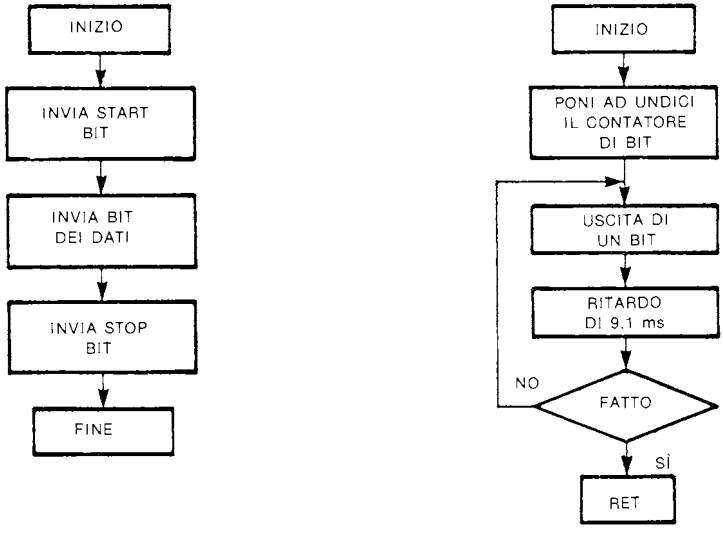


Figura 6-16: Output su Telescrivente

Esercizio 6-25: *Scrivete un programma di ritardo che realizzi il ritardo di 9.09 millisecondi (subroutine di DELAY).*

Esercizio 6-26: *Usando l'esempio del programma sviluppato sopra, scrivete un programma PRINTC che stampi su Telescrivente il contenuto della posizione di memoria CHAR (vedere la Figura 6-15).*

La risposta è riportata di seguito:

```
PRINTC   LD     B, 11          CONTATORE=11 BIT
         LD     A, (CHAR)      ACCETTA CARATTERE
         OR     A              AZZERA IL RIPORTO=START BIT
         RLA                   RIPORTO IN A
NEXT     OUT    (TTYBIT), A    OUTPUT
         CALL   DELAY
         RRA                   PROSSIMO BIT
         SCF                   RIPORTO=1 (STOP BIT)
         DEC    B              CONTEGGIO DI BIT
         JR     NZ, NEXT
         RET
```

Il registro B è usato come contatore di bit per la trasmissione. Il contenuto del Bit 0 di A sarà inviato alla linea della Telescrivente (TTYBIT). Notate come viene impiegato il riporto per fornire un nono bit (lo START Bit).

Notate anche che il riporto è azzerato con:

```
         OR     A
```

Alla fine del programma, il riporto è posto ad uno da:

```
         SCF
```

per generare un bit d'arresto.

Esercizio6-27: *Modificate il programma cosicché aspetti uno START bit invece di uno STATUS bit.*

Stampa di una stringa di caratteri

Supporremo che la routine PRINTC (vedere Esercizio 6-26) consenta di visualizzare un carattere sulla nostra stampante, o display, oppure qualsiasi dispositivo d'output. Qui stamperemo il contenuto delle posizioni di memoria da (START) a (START + N). Il programma è semplice (vedere la Figura 6-17):

```
PSTRING  LD     B, NBR         LUNGHEZZA DELLA STRINGA
         LD     HL, START      INDIRIZZO DI BASE
NEXT     LD     A, (HL)        PRENDI IL CARATTERE
         CALL   PRINTC         STAMPALO
         INC    HL             PROSSIMO ELEMENTO
         DEC    B
         JR     NZ, NEXT       RIPETI
         RET
```

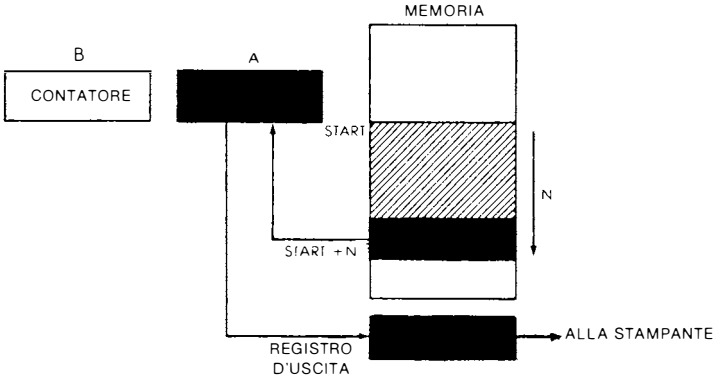


Figura 6-17: Stampa di un blocco di memoria

SOMMARIO SULLE PERIFERICHE

Abbiamo descritto le tecniche fondamentali della programmazione usate per comunicare con dispositivi d'input/output tipici.

Oltre al trasferimento dei dati, sarà necessario condizionare uno o più registri di controllo all'interno di ogni dispositivo I/O per condizionare correttamente le velocità di trasferimento, il meccanismo d'interrupt e le altre varie opzioni. Dovrebbe essere consultato il manuale per ogni dispositivo. (Per altri dettagli sugli algoritmi specifici per scambiare informazioni con tutti i periferici più comuni, il lettore si può riferire al nostro libro, C207, *Tecniche di Interfacciamento dei Microprocessori*).

Abbiamo imparato a governare dispositivi singoli. Comunque, in un sistema reale, tutti i periferici sono collegati ai bus e possono richiedere servizio simultaneamente. Come abbiamo intenzione di programmare il tempo del processore?

SCHEDULING D'INPUT/OUTPUT

Poiché le richieste d'input/output possono avvenire simultaneamente, deve essere realizzato un meccanismo di scheduling (programmazione o inventario) in ogni sistema per determinare in quale ordine sarà concesso il servizio. Sono usate tre tecniche fondamentali d'input/output, le quali possono essere combinate in tutti i modi possibili.

Sono: polling, interrupt, DMA. Il polling e gli interrupt saranno descritti qui. Il DMA è una tecnica puramente hardware e come tale non sarà descritto qui. (Sarà trattato nei libri di riferimento C201 e C207).

Polling

Concettualmente, il polling (registrazione) è il metodo più semplice per trattare i periferici multipli. Con questa strategia, il processore interroga alternativamente i dispositivi collegati ai bus. Se un dispositivo richiede servizio, il servizio è concesso. Se non richiede servizio viene esaminato il periferico successivo. Il polling è usato non solo per i dispositivi, ma per ogni *routine di servizio di dispositivo*.

Come esempio, se il sistema è fornito di una Telescrivente, di un registratore e di un display CRT, la polling routine interrogherebbe la Telescrivente: "Hai un carattere da trasmettere?" Interrogherebbe la *routine d'output* della Telescrivente, chiedendo: "Hai un carattere da inviare?". Poi, supponendo che fino a questo momento le risposte siano negative, interrogherebbe la routine del registratore e alla fine interrogherebbe il display CRT. Se solo un dispositivo è collegato ad un sistema, il polling sarà usato per determinare se ha bisogno di servizio. Come esempio, i diagrammi di flusso per leggere da un dispositivo di lettura a nastro di carta e per stampare su una stampatrice compaiono nelle Figure 6-20 e 6-21.

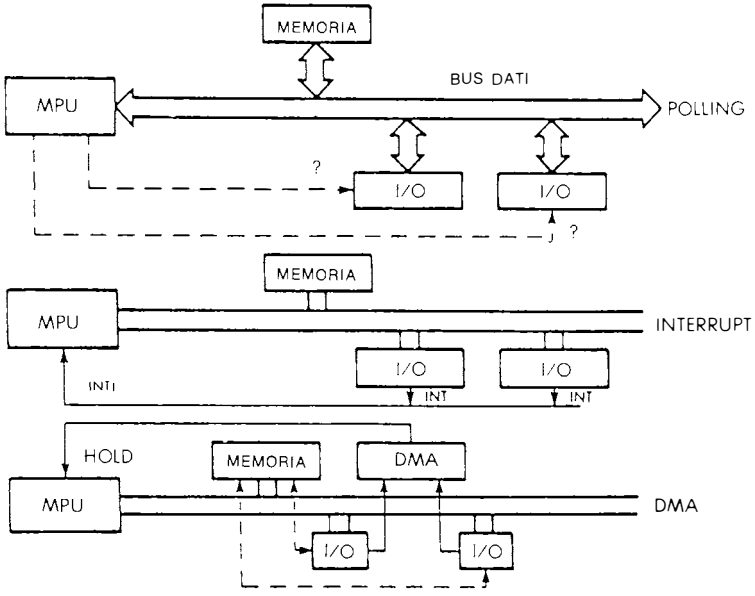


Figura 6-18: Tre metodi di controllo I/O

Esempio: un loop di polling per dispositivi 1, 2, 3, 4 (vedere la Figura 6-19):

```
POLL4   IN      A, (STATUS 1)   ACCETTA LO STATO DEL DISPOSITIVO 1
        BIT     7, A            RICHIESTA DI SERVIZIO
        CALL    NZ, ONE         BIT 7 = 1?
        IN      A, (STATUS 2)   DISPOSITIVO 2
        BIT     7, A
        CALL    NZ, TWO
        IN      A, (STATUS 3)   DISPOSITIVO 3
        BIT     7, A
        CALL    NZ, THREE
        IN      A, (STATUS 4)   DISPOSITIVO 4
        BIT     7, A
        CALL    NZ, FOUR
        JR      POLL4           NESSUNA RICHIESTA, RIPROVA
```

Il bit 7 del registro di stato per ogni dispositivo è "1" quando vuole servizio. Quando è percepita una richiesta, questo programma produce una ramificazione al manipolatore dei dispositivi all'indirizzo ONE per il dispositivo 1, TWO per il dispositivo 2, ecc.

Qui è degno di essere notato un particolare acuto. Per ogni istruzione, è importante verificare attentamente il modo in cui interessa i codici di condizione. Si dovrebbe notare che l'istruzione d'input non varia i flag. Se è stata usata un'istruzione per caricamento di memoria invece di una istruzione d'input, il bit 7 dell'input è automaticamente riflesso come il SIGN bit nel registro dei flag. L'istruzione speciale "BIT 7, A" diverrebbe inutile. Comunque, questo test extra deve essere incluso nel programma perché l'istruzione IN non varia i flag.

In alcune strutture hardware, i dispositivi d'input/output possono essere trattati come dispositivi di memoria per scopi d'indirizzamento. Questo è chiamato input/output in mappaggio di memoria (memory-mapped I/O). In questo caso, l'istruzione IN sarebbe sostituita da una istruzione LD ed il resto del programma sarebbe modificato come indicato sopra.

I vantaggi del polling sono ovvi: è semplice, non richiede nessuna assistenza hardware, e mantiene sincroni tutti gli input/output con il funzionamento del programma. Il suo svantaggio è altrettanto ovvio: la maggior parte del tempo del processore è consumato a guardare dispositivi che non hanno bisogno di servizio. Inoltre, consumando così tanto tempo, il processore potrebbe dare servizio ad un dispositivo troppo tardi. Perciò, è desiderabile un altro meccanismo per garantire che il tempo del processore può essere usato per eseguire calcoli utili piuttosto che fare del polling senza necessità per tutto il tempo. Comunque, sottolineiamo che il polling è usato in modo esteso ogni qual volta un microprocessore non ha niente di meglio da fare, siccome mantiene semplice l'organizzazione globale. Esaminiamo l'alternativa essenziale al polling: interrupt.

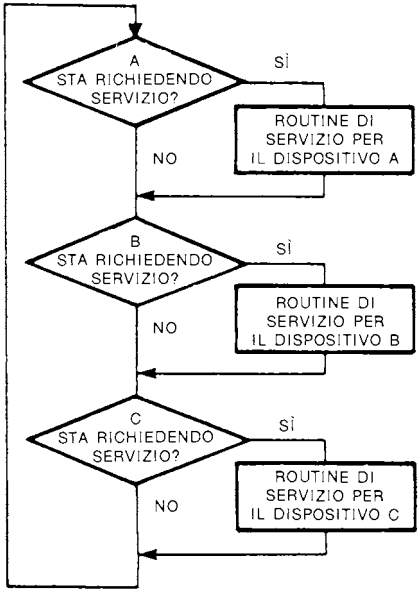


Figura 6-19: Diagramma di Flusso del Ciclo di Polling

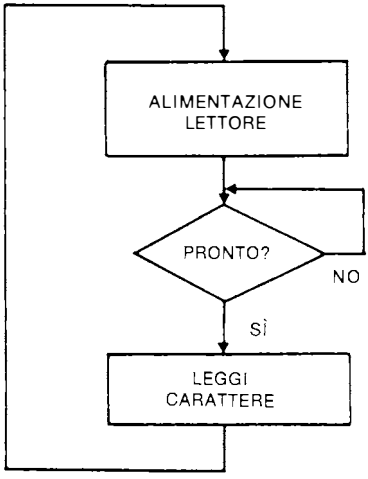


Figura 6-20: Lettura da un lettore di nastro perforato

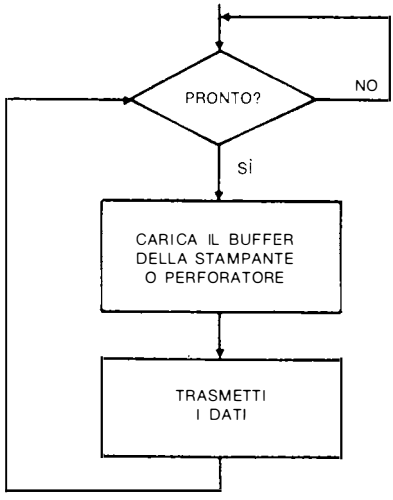


Figura 6-21: Stampa su Perforatore o Stampante

Interrupt

Il concetto di interrupt è illustrato nella Figura 6-18. Una linea hardware speciale, la linea d'interrupt, è collegata ad un pin specializzato del microprocessore. Dispositivi d'input/output multipli possono essere collegati a questa linea d'interrupt. Quando qualcuno di essi ha bisogno di servizio, invia un livello o un inpulso su questa linea. Un segnale d'interrupt è il servizio richiesto da un dispositivo d'input/output al processore. Esaminiamo la risposta del processore a questo interrupt.

In ogni caso, il processore completa l'istruzione che stava correttamente eseguendo; in caso contrario, questo creerebbe del caos all'interno del microprocessore stesso. Quindi il microprocessore salta ad una routine che gestisce l'interrupt la quale elaborerà l'interrupt. La ramificazione verso una tale subroutine implica che il contenuto del contatore di programma PC deve essere salvato sullo stack. *Perciò, un interrupt deve causare la conservazione automatica del PC sullo stack.* Inoltre, anche il registro di flag F dovrebbe essere conservato automaticamente, siccome il suo contenuto sarà alterato da qualsiasi istruzione successiva.

Alla fine, se la routine che gestisce l'interrupt dovesse modificare dei registri interni, anche tali registri interni dovrebbero essere preservati sullo stack (vedere le Figure 6-22 e 6-23).

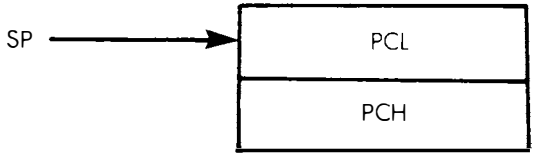


Figura 6-22: Lo stack dello Z80 dopo l'interruzione

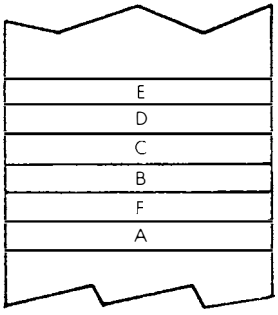


Figura 6-23: Salvataggio di alcuni registri operativi

Si può saltare all'appropriato indirizzo che gestisce l'interrupt dopo che sono stati conservati tutti questi registri. Alla fine di questa routine, dovrebbero essere ripristinati tutti i registri, e dovrebbe essere eseguito un ritorno speciale da interrupt in modo che il programma principale riprenda l'esecuzione. Ora esaminiamo più dettagliatamente le linee d'interrupt dello Z80.

Interrupt per lo Z80

Un interrupt è un segnale inviato al microprocessore che può richiedere servizio ad ogni momento ed è asincrono al programma. Ogni qual volta un programma si ramifica verso una subroutine, tale ramificazione è *sincrona* alla esecuzione del programma, cioè, programmata dal programma. Comunque, può accadere un interrupt ad ogni momento e generalmente sospenderà l'esecuzione del programma corrente (senza che lo sappia il programma). È chiamato *asincrono* perché può capitare in qualsiasi istante durante l'esecuzione del programma.

Sullo Z80 sono forniti tre meccanismi d'interruzione: la richiesta di bus (BUSRQ), l'interrupt non mascherabile (NMI) e l'interrupt comune (INT).

Esaminiamo questi tre tipi.

La richiesta di bus

La richiesta di bus è il meccanismo d'interrupt a più alta priorità sullo Z80. Nella Figura 6-24 è mostrata la sequenza d'interrupt per lo Z80. Come regola generale, non sarà percepito nessun interrupt dallo Z80 fino a che è completato il ciclo di macchina corrente. Gli interrupt NMI e INT non saranno presi in considerazione fino a quando non è terminata l'istruzione corrente. Comunque, il BUSRQ sarà manipolato alla fine del ciclo di macchina corrente, senza necessariamente aspettare la fine dell'istruzione. È usato per un diretto accesso alla memoria (DMA), e farà andare lo Z80 nel modo DMA (vedere riferimento al C201 per una spiegazione del meccanismo DMA). Se è stata raggiunta la fine di una istruzione, e se l'NMI oppure l'INT fossero pendenti, sarebbero memorizzati internamente nello Z80 tramite la regolazione flip-flop specializzati: il flip-flop NMI, e il flip-flop INT. Nel modo DMA, lo Z80 sospende il funzionamento e

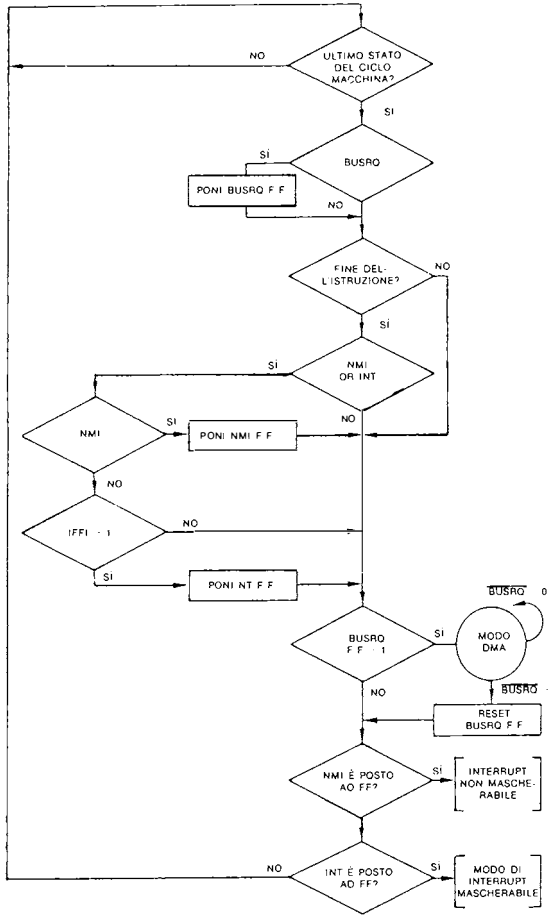


Figura 6-24: Sequenza d'interrupt

pone il bus dei dati e quello degli indirizzi nello stato di alta impedenza. Questo modo è normalmente usato da un controllore DMA per eseguire trasferimenti tra un dispositivo d'input/output ad alta velocità e la memoria, usando il bus dei dati e il bus degli indirizzi del microprocessore. La fine del funzionamento DMA è indicata allo Z80 dalla variazione dei livelli BUSRQ. A questo punto, lo Z80 riprenderà il funzionamento normale. In particolare, verificherà se il suo flip-flop NMI o INT interno era stato posto in condizione di set e, se lo è stato, eseguirà gli interrupt corrispondenti.

Il DMA non dovrebbe essere normalmente preoccupante per il programmatore, a meno che sia importante la temporizzazione. Se è presente un controllore DMA nel sistema, il programmatore deve capire che il DMA può ritardare la risposta ad un NMI oppure ad un INT.

L'interrupt non mascherabile

Questo tipo di interrupt non può essere inibito dal programmatore. È perciò detto essere *non mascherabile*, e da questo deriva il suo nome. Sarà sempre accettato dallo Z80 sul compimento dell'istruzione corrente, supponendo che non sia stata ricevuta nessuna richiesta di bus. (Se un NMI è ricevuto durante un BUSRQ, porrà in condizione di set il flip-flop NMI interno, e sarà elaborato alla fine del BUSRQ).

NMI provocherà un trasferimento automatico del contatore di programma nello stack ed una ramificazione ad un *vettore d'interrupt* a 16 bit, all'indirizzo 0066H: i due byte che rappresentano l'indirizzo immagazzinato alla posizione 0066H saranno caricati nel contatore di programma; essi rappresentano l'indirizzo di inizio della routine di gestione per l'NMI (vedere la Figura 6-25).

Questo meccanismo d'interrupt è stato progettato per alta velocità, poichè è usato nel caso di emergenze. Perciò, non offre la flessibilità del modo d'interrupt mascherabile, descritto sotto.

Notate che un vettore d'interrupt deve essere stato caricato all'indirizzo 0066H prima di usare l'NMI.

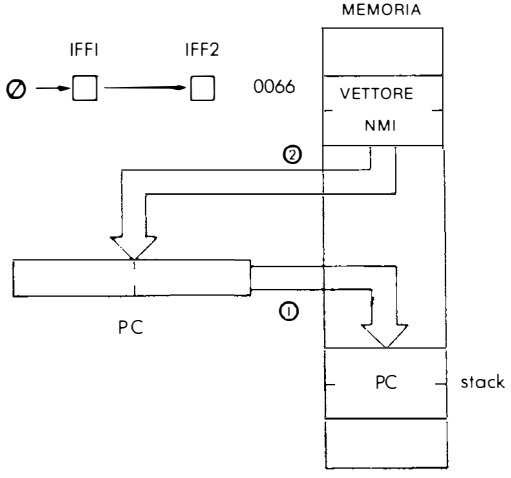


Figura 6-25: NMI Forza la Vettorizzazione Automatica

Un "vettore d'interrupt" è semplicemente un indirizzo o un altro puntatore che specifica la posizione di memoria della routine di manipolazione interrupt (interrupt-handling routine). In questo caso, il meccanismo è un *accesso di memoria indiretto*. L'hardware ramifica sempre a 0066H, ma il programmatore è libero di caricare un qualche indirizzo di vettorizzazione in questa posizione di memoria.

Poi, lo stato del flip-flop del bit maschera dell'interrupt (IFF1) all'istante in cui è stato ricevuto NMI è conservato automaticamente in IFF2. Quindi si ha il reset di IFF1 per impedire qualsiasi altro interrupt. Questa caratteristica è importante per impedire la perdita di INT di più bassa priorità e semplifica l'hardware esterno: lo stato di un INT pendente è conservato internamente nello Z80. L'interrupt NMI è normalmente usato per eventi ad alta priorità come un clock in tempo reale oppure una caduta dell'alimentazione.

Il ritorno da un NMI è compiuto da una istruzione speciale, RETN: "ritorno da un interrupt non mascherabile". Il contenuto di IFF1 è ristabilito da IFF2 ed il contenuto del contatore di programma PC è ristabilito dalla sua posizione nello stack. Dal momento in cui si è avuto il reset di IFF1, durante l'esecuzione dell'NMI, nessun INT esterno poteva essere accettato durante l'NMI: non c'è stata nessuna perdita di informazioni.

Interrupt

L'interrupt INT ordinario e mascherabile può funzionare in tre modi. Sono specifici allo Z80, poiché l'8080 è fornito soltanto di un solo modo d'interrupt. L'interrupt INT ordinario può anche essere mascherato selettivamente dal programmatore. Ponendo i flip-flop d'interrupt IFF1 e IFF2 ad un "1", autorizzerà gli interrupt. Ponendoli a "0" (mascherandoli) impedirà la rivelazione di INT. L'istruzione EI è usata per porli ad 1, e l'istruzione DI è usata per porli a 0. IFF1 e IFF2 sono simultaneamente posti ad 1 ed a 0. Durante l'esecuzione delle istruzioni Ei e DI, gli INT sono disabilitati per prevenire una qualche perdita d'informazioni.

Esaminiamo i tre modi d'interrupt:

Modo d'interrupt 0

Questo modo è identico al modo d'interrupt dell'8080. Lo Z80 opererà nel modo d'interrupt 0 all'atto dell'accensione iniziale (quando è stato applicato il segnale RESET) oppure quando è stata eseguita una istruzione IM0. Una volta predisposto il modo 0 sarà riconosciuto un interrupt se il flip-flop che abilita l'interrupt IFF1 è posto ad 1, purché allo stesso tempo non avvenga nessuna richiesta di bus o un interrupt non mascherabile. L'interrupt sarà rivelato soltanto alla fine di una istruzione.

Essenzialmente, lo Z80 risponderà all'interrupt generando un IORQ (cosi come un segnale M1), e poi non fa altro che aspettare.

È responsabilità di un *dispositivo esterno* riconoscere l'IORQ oppure l'M1 (questo è chiamato un *riconoscimento d'interrupt* o INTA) e porre una istruzione sul bus dei dati. Lo Z80 aspetta che un'istruzione sia posta sul suo bus dei dati dal dispositivo esterno entro il prossimo ciclo. Tipicamente, sul bus è posta una istruzione RST o CALL. Entrambe queste due istruzioni conservano automaticamente il program counter nello stack, e provocano la ramificazione ad un indirizzo specifico. Il vantaggio dell'istruzione RST è che risiede in un byte singolo, cioè, viene eseguita rapidamente. Lo svantaggio è di ramificare a solo una delle otto posizioni possibili nella pagina zero (indirizzi da 0 a 225). Il vantaggio dell'istruzione CALL è quella di essere una istruzione di ramificazione a scopo generale che specifica un indirizzo a 16 bit completo. Comunque, richiede tre byte e perciò viene eseguita meno rapidamente.

Notate che una volta che inizia l'elaborazione di interrupt, tutti gli ulteriori interrupt sono disabilitati. IFF1 e IFF2 sono automaticamente posti a "0". Allora è responsabilità del programmatore inserire una istruzione EI (abilitazione interrupt) in posizione appropriata all'interno del suo programma se desidera abilitare gli interrupt, e, in ogni caso, prima di ritornare dall'interrupt.

Nella Figura 6-26 è mostrata la sequenza dettagliata che corrisponde al modo d'interrupt 0.

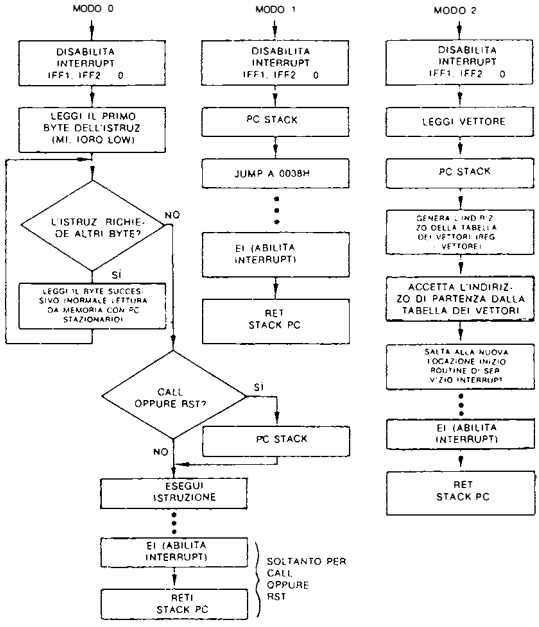


Figura 6-26: Modi di interrupt

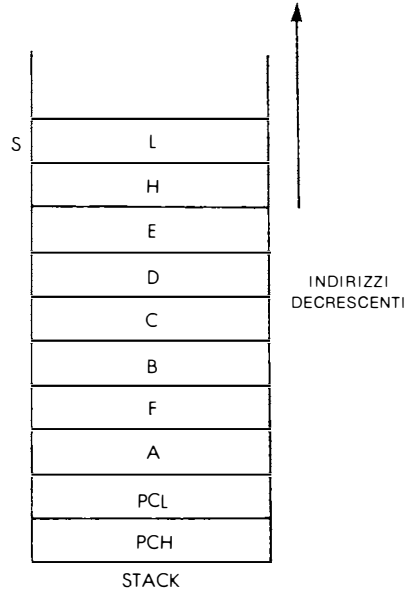


Figura 6-27: Salvataggio dei Registri

Il ritorno dall'interrupt è compiuto tramite un'istruzione RETI. A questo punto ricordiamo al programmatore che è lui di solito responsabile per cancellare esplicitamente l'interrupt che è stato servito sul dispositivo I/O, e sempre per ripristinare il flag che disabilita l'interrupt all'interno dello Z80. Comunque, il controllore di periferica può usare il segnale INTA per cancellare la richiesta INT, liberando così il programmatore da questo lavoro.

Inoltre, se la routine che manipola l'interrupt dovesse modificare il contenuto di qualcuno dei registri interni, il programmatore è specificatamente responsabile della conservazione di questi registri nello stack prima di eseguire la routine che manipola l'interrupt. Altrimenti, il contenuto di questi registri sarà distrutto e, quando il programma interrotto riprende l'esecuzione, non funzionerà. Per esempio, supponendo che i registri A, B, C, D, E, H ed L siano usati all'interno del manipolatore d'interrupt, dovranno essere salvati (vedere la Figura 6-27).

Il programma corrispondente è:

```
SAVREG   PUSH   AF
         PUSH   BC
         PUSH   DE
         PUSH   HL
```

Questi registri devono essere ripristinati dopo il compimento della routine che manipola l'interrupt. La gestione dell'interrupt terminerà con la seguente sequenza d'istruzioni:

```
         POP    HL
         POP    DE
         POP    BC
         POP    AF
         EI              (a meno che EI compaia precedentemente nella routine)
```

Inoltre, se i registri IX e IY sono usati dalla routine, anch'essi devono essere conservati e poi richiamati.

Modo d'interrupt 1

Questo modo d'interrupt viene selezionato eseguendo l'istruzione IM1. È un manipolatore d'interrupt automatizzato che provoca una ramificazione automatica alla posizione 0038H.

È perciò essenzialmente analogo al meccanismo d'interrupt NMI eccetto il fatto che può essere mascherabile. Lo Z80 conserva automaticamente il contenuto di PC nello stack (vedere la Figura 6-28).

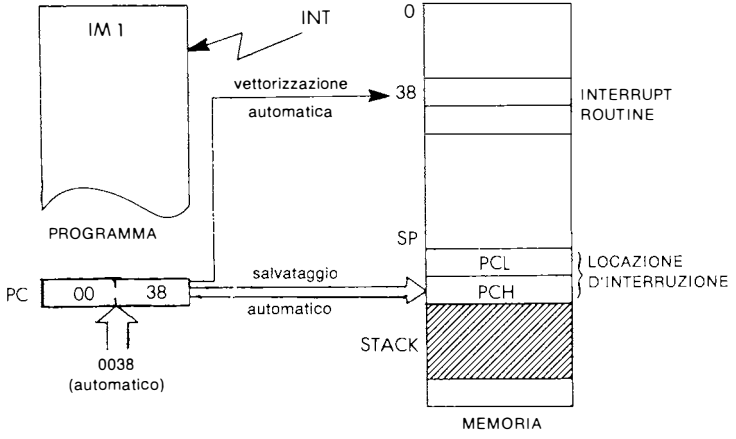


Figura 6-28: Interrupt — Modo 1

Questa risposta d'interrupt automatizzato, la quale riporta tutti gli interrupt alla posizione di memoria 38H, minimizza la quantità di hardware necessaria per usare gli interrupt. Il suo svantaggio possibile è quello di causare una ramificazione ad una posizione di memoria *singola*. Nel caso siano collegati diversi dispositivi alla linea INT, il programma che inizia alla posizione 38H sarà responsabile per determinare quale dispositivo richiedeva assistenza. Questo problema sarà trattato in seguito.

Per quanto riguarda la temporizzazione di questo interrupt deve essere presa una precauzione: quando si eseguono trasferimenti input/output programmati, lo Z80 ignorerà qualsiasi dato che può essere presente nel bus dei dati durante il ciclo che segue l'interrupt (il ciclo di riconoscimento interrupt).

Modo d'interrupt 2 (Interrupt Vettorizzati)

Questo modo è inserito eseguendo una istruzione IM2 (Interrupt Mode 2). È un modo potente che permette la vettorizzazione automatica degli interrupt. Il vettore interrupt è un indirizzo fornito dal dispositivo periferico che genera l'interrupt, e usato come un puntatore di memoria all'indirizzo d'inizio della routine che gestisce l'interrupt. Il meccanismo d'indirizzamento fornito dallo Z80 nel modo 2 è indiretto, anziché diretto. Ogni periferico fornisce un indirizzo di ramificazione di sette bit che è aggiunto all'indirizzo ad 8 bit contenuto nello speciale registro I dello Z80. Il bit più a destra dell'indirizzo a 16 bit è posto a "0". Questo indirizzo risultante punta ad un ingresso in una tabella da qualche parte nella memoria. Questa tabella può contenere fino ad otto ingressi costituiti da parole doppie. Ognuna di queste parole doppie è l'indirizzo del manipolatore d'interrupt per il dispositivo corrispondente. Questo è illustrato nelle Figure 6-29 e 6-30.

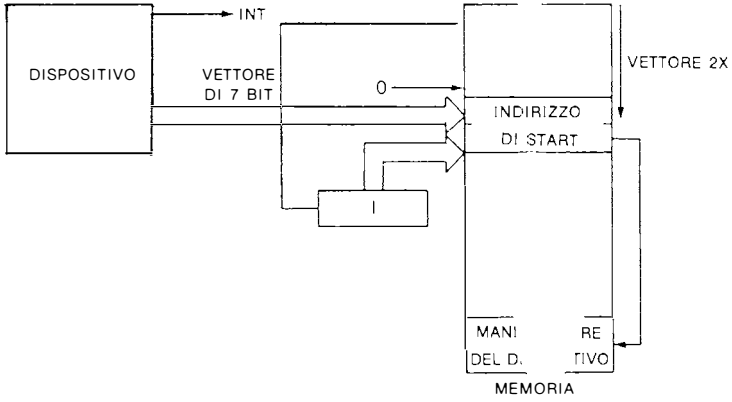


Figura 6-29: Interrupt — Modo 2

La tabella di interrupt può avere fino a 128 ingressi. In questo modo, lo Z80 spinge automaticamente il contenuto del contatore di programma nello stack. Ciò è ovviamente necessario, dal momento che il PC sarà ricaricato con il contenuto dell'ingresso della tabella d'interrupt che corrisponde al vettore fornito dal dispositivo.

"Overhead" (lavoro addizionale) introdotto dall'interrupt

Riferitevi alla Figura 6-18 per un confronto grafico del processo di polling rispetto il processo d'interrupt, dove il processo di polling è illustrato in alto, e il processo d'interrupt in basso.

Si può vedere che nella tecnica di polling il programma consuma molto tempo di attesa.

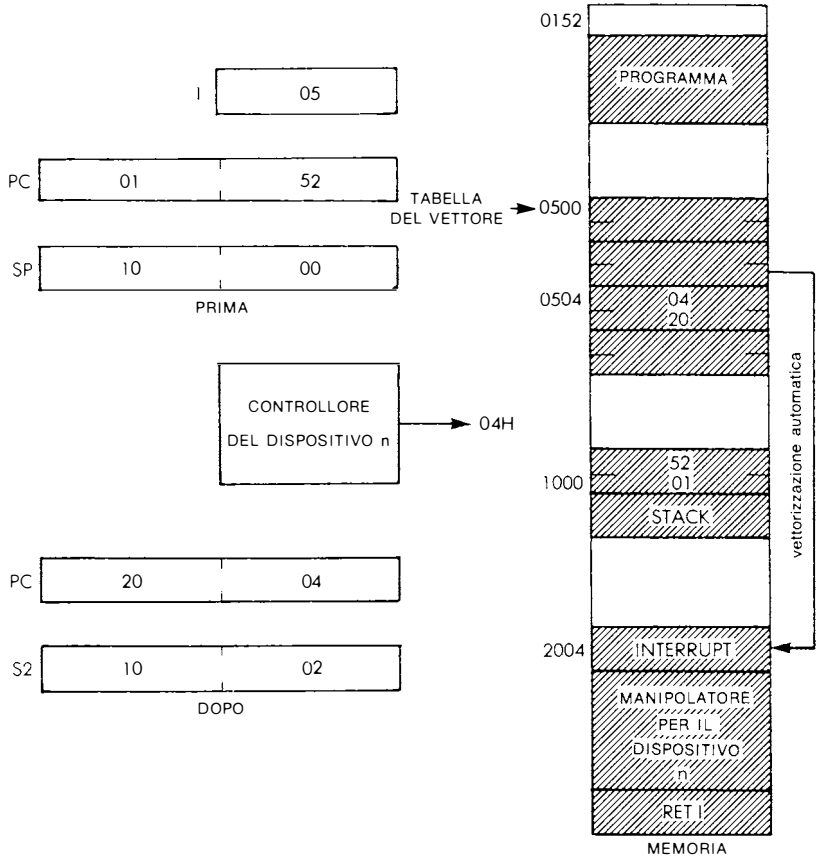


Figura 6-30: Modo 2 — Un esempio pratico

Usando gli interrupt, il programma è interrotto, l'interrupt è eseguito e poi il programma viene ripreso. Comunque, l'ovvio svantaggio di un interrupt è quello di introdurre diverse istruzioni addizionali all'inizio e alla fine, con la conseguente introduzione di un ritardo prima che possa essere eseguita la prima istruzione del manipolatore del dispositivo.

Questo è un "overhead" addizionale.

Esercizio 6-28: *Usando la tabella nell'Appendice indicante il numero di cicli per istruzione, calcolate quanto tempo sarà impiegato per salvare e poi richiamare i registri A, B, D, H.*

Avendo chiarito il funzionamento delle linee di interrupt, adesso consideriamo due importanti problemi rimanenti:

1 — Come risolviamo il problema dei dispositivi multipli che fanno scattare un interrupt allo stesso tempo?
2 — Come risolviamo il problema di un interrupt che avviene mentre sta per essere eseguito un altro interrupt?

Dispositivi multipli collegati ad una linea d'interrupt singola

Ogni qual volta avviene un interrupt, il processore si ramifica ad un indirizzo specificato. Prima che faccia una qualsiasi elaborazione effettiva, la routine che gestisce l'interrupt deve determinare quale dispositivo ha fatto scattare l'interrupt. Come al solito, sono disponibili due metodi per identificare il dispositivo: un metodo software e un metodo hardware.

Nel metodo software è usato il polling: il microprocessore alternativamente interroga ognuno dei dispositivi e gli chiede, "Hai fatto scattare l'interrupt?".

Se la risposta è negativa, interroga quello successivo. Questo processo è illustrato nella Figura 6-31. Un programma campione è:

```
POLINT   IN    A, (STATUS 1)   LEGGI LO STATO
         BIT   7, A            IL DISPOSITIVO HA RICHIESTO L'INT?
         JP    NZ, ONE         SE SÌ, GESTISCILO
         IN    A, (STATUS 2)
         BIT   7, A
         JP    NZ, TWO
         ecc.
```

Il metodo hardware usa componenti addizionali ma fornisce l'indirizzo del dispositivo d'interruzione simultaneamente con la richiesta d'interrupt. Il dispositivo ora universalmente usato per fornire questo servizio è chiamato "PIC", o controllore della priorità dell'interrupt. Tale PIC porrà automaticamente sul bus dei dati l'indirizzo di ramificazione realmente richiesto per il periferico che interrompe. Per essere più precisi, quando si opera nel modo 0, il PIC fornirà un RST di un byte oppure una CALL di tre byte sul bus dei dati in risposta al riconoscimento d'interrupt, automatizzando perciò la vettorizzazione di interrupt, e minimizzando l'overhead.

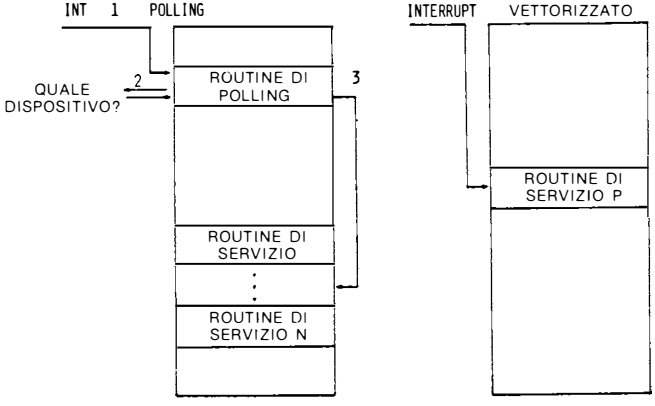


Figura 6-31: Il Polled Interrupt rispetto all'Interrupt vettorizzato

Notate che è richiesta una istruzione di chiamata di subroutine poichè lo Z80 non salva il PC quando sta operando nel modo 0.

Nella maggioranza dei casi, la velocità di reazione ad un interrupt non è cruciale ed è usato un approccio polling. Deve essere usato un approccio hardware se il tempo di risposta è una considerazione primaria.

Interrupt simultanei

Un altro problema che può capitare è quello che un nuovo interrupt può essere fatto scattare durante l'esecuzione di una routine che gestisce l'interrupt. Esaminiamo cosa accade e come è usato lo stack per risolvere il problema. Nel Capitolo 2 abbiamo indicato che questo era un altro ruolo essenziale dello stack, ed ora è giunto il momento di dimostrare il suo uso. Per illustrare gli interrupt multipli, ci riferiremo alla Figura 6-33. Nell'illustrazione il tempo va da sinistra a destra. Il contenuto dello stack è mostrato nella parte bassa dell'illustrazione. Guardando a sinistra, al tempo 0, il programma P è in esecuzione. Successivamente al tempo T1, avviene l'interrupt I1. Supporremo che la maschera d'interrupt sia stata abilitata, autorizzando I1. Il programma P sarà sospeso. Questo è mostrato nella parte bassa dell'illustrazione. Lo stack conterrà almeno il program counter ed il registro di stato del programma P, più qualsiasi altro registro opzionale che potrebbe essere salvato dal manipolatore di interrupt o dallo stesso I1.

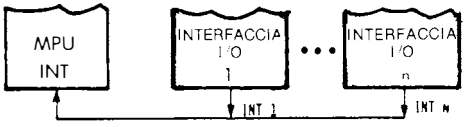


Figura 6-32: Più dispositivi possono usare la stessa linea d'Interrupt

Al tempo T1, l'interrupt I1 viene eseguito fino al tempo T2. Al tempo T2, avviene l'interrupt I2. Supporremo che l'interrupt I2 abbia una priorità più alta dell'interrupt I1. Se aveva una priorità più bassa, sarebbe stato ignorato fino a quando non fosse stato completato I1. Al tempo T2, i registri per I1 sono messi sullo stack e ciò compare nella parte bassa dell'illustrazione. Il contenuto del program counter ed AF sono spinti di nuovo nello stack. Inoltre, la routine per I2 potrebbe richiedere di salvare alcuni registri addizionali. Adesso I2 verrà eseguito fino al completamento al tempo T3.

Quando termina I2, il contenuto dello stack è automaticamente riprelevato nello Z80, e questo è mostrato alla fine della Figura 6-33. Così, I1 riprende l'esecuzione automaticamente. Sfortunatamente, al tempo T4, avviene di nuovo un interrupt I3 a priorità più alta. Alla fine dell'illustrazione possiamo vedere che i registri per I1 sono di nuovo spinti nello stack. L'interrupt I3 viene eseguito da T4 a T5 e termina a T5.

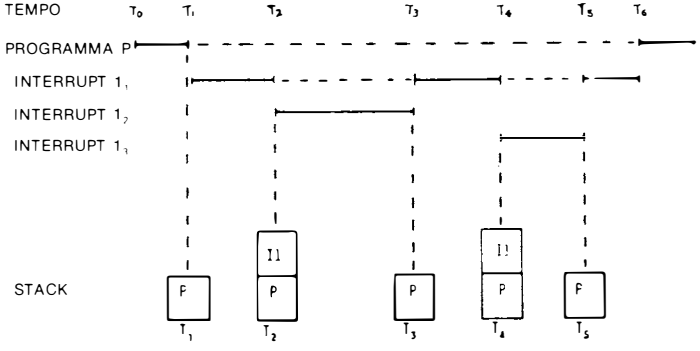


Figura 6-33: Contenuti dello Stack durante Interrupt multipli

In quel momento, il contenuto dello stack è prelevato nello Z80 e l'interrupt I1 riprende l'esecuzione. Questa volta viene eseguito fino al completamento e termina a T6. A T6, i rimanenti registri che sono stati salvati nello stack sono prelevati nello Z80 e il programma P può riprendere l'esecuzione.

A questo punto il lettore verificherà che lo stack sia vuoto. Infatti, il numero di linee tratteggiate che indicano la sospensione del programma indicano allo stesso tempo quanti livelli ci sono nello stack.

Esercizio 6-29: *Supponiamo che l'area disponibile per lo stack sia limitata a 300 locazioni in un programma specifico. Supponiamo che tutti i registri debbano essere sempre salvati e che il programmatore permetta che gli interrupt siano annidati, cioè uno dentro l'altro. Qual è il numero massimo di interrupt simultanei che può essere manipolato? Un qualsiasi altro fattore contribuirà a ridurre ancora di più il numero massimo d'interrupt simultanei?*

Comunque, in pratica, deve essere sottolineato che i sistemi di microprocessori sono normalmente collegati ad un piccolo numero di dispositivi che usano gli interrupt. Perciò, è improbabile che in un tale sistema avvenga un numero elevato d'interrupt simultanei.

Adesso abbiamo risolto tutti i problemi di solito associati con gli interrupt. Infatti, il loro uso è semplice e dovrebbero essere impiegati con profitto anche dal programmatore principiante.

SOMMARIO

In questo capitolo abbiamo presentato la gamma delle tecniche usate per comunicare con il mondo esterno. Da routine d'input/output elementari a programmi più complessi per comunicare con periferici reali, abbiamo imparato a sviluppare tutti i programmi comuni ed abbiamo esaminato anche l'efficienza dei programmi benchmark nel caso di un trasferimento parallelo e di una conversione parallelo - serie.

Alla fine, abbiamo imparato a organizzare il funzionamento dei periferici multipli usando il polling e gli interrupt. Naturalmente ad un sistema potrebbero essere collegati molti altri dispositivi d'input/output. Dovrebbe essere possibile risolvere i problemi più comuni con la serie di tecniche che sono state presentate fino ad ora e con la comprensione dei periferici connessi.

Nel prossimo capitolo, esamineremo le caratteristiche reali dei chip d'interfaccia d'input/output di solito collegati ad uno Z80. Poi, considereremo le strutture dati fondamentali che il programmatore può usare.

Esercizio 6-30: *Calcolate l'overhead quando operate nel modo 0, supponendo che tutti i registri siano salvati, e che sia ricevuto un RST in risposta al riconoscimento interrupt. L'overhead è definito come il ritardo totale avvenuto, esclusivo delle istruzioni richieste, per rendere effettivo il processo d'interrupt propriamente detto.*

Esercizio 6-31: *Un display LED a 7 segmenti può anche mostrare digit diversi dall'alfabeto esadecimale. Calcolate i codici per: H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, q, r, t, u, y.*

Esercizio 6-32: *Nella Figura 6-34 compare il diagramma di flusso per la gestione dell'interrupt. Rispondete alle seguenti domande:*

a — Cosa è fatto via hardware e cosa è fatto via software?
b — Qual è l'uso della maschera?
c — Quanti registri dovrebbero essere conservati?
d — Com'è identificato il dispositivo d'interruzione?
e — Cosa fa l'istruzione RETI? Come differisce da un ritorno di subroutine?
f — Suggerite un modo per trattare una situazione di overflow dello stack.
g — Cos'è l'overhead ("tempo perso") introdotto dal meccanismo d'interrupt?

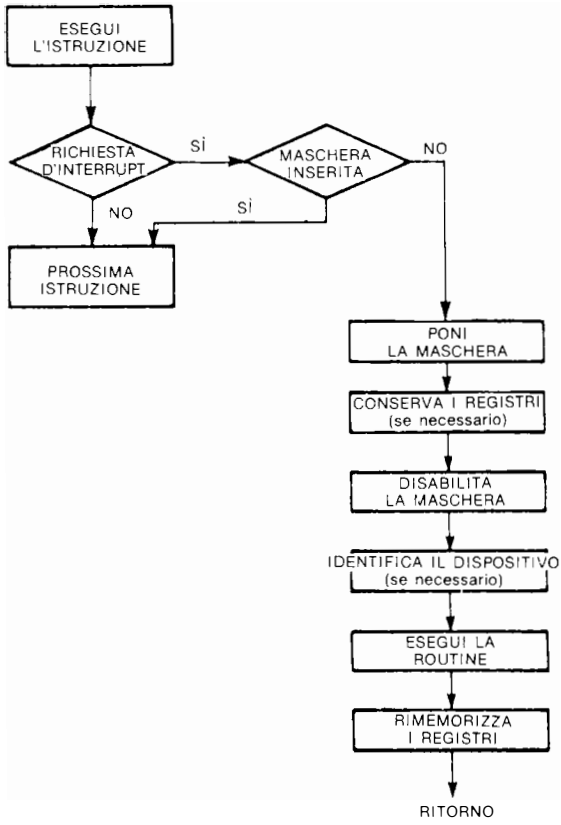


Figura 6-34: Logica dell'Interrupt

CAPITOLO 7

DISPOSITIVI D'INPUT/OUTPUT

INTRODUZIONE

Abbiamo imparato come programmare il microprocessore Z80 nelle situazioni più comuni. Comunque, dovremmo fare una trattazione speciale dei chip d'input/output normalmente collegati al microprocessore. A causa del progresso nell'integrazione LSI, sono stati introdotti nuovi chip che non esistevano prima. Come risultato, programmare un sistema richiede, naturalmente, per prima cosa di programmare il microprocessore stesso, e poi *programmare i chip d'input/output*. Infatti, è spesso più difficile ricordare come programmare le varie funzioni di controllo di un chip d'input/output che programmare il microprocessore stesso! E questo non perché la programmazione in se stessa sia più difficile, ma perché ognuno di questi dispositivi ha le sue proprie idiosincrasie. Qui per prima cosa esamineremo il dispositivo d'input/output più generale, il chip d'input/output programmabile ("PIO") e poi alcuni dispositivi I/O Zilog.

Il "PIO Standard"

Non c'è nessun "PIO standard". Comunque, ogni dispositivo PIO è essenzialmente analogo nella funzione a tutti i dispositivi PIO prodotti per lo stesso scopo da altri fabbricanti. Lo scopo di un PIO è quello di fornire un collegamento multiporta per i dispositivi d'input/output. (Una "porta" è semplicemente un set di 8 linee d'input/output). Ogni PIO fornisce almeno due set di linee ad 8 bit per i dispositivi d'I/O. Ogni dispositivo d'I/O ha bisogno di un *"data buffer"* per stabilizzare il contenuto del bus dei dati almeno sull'output. Perciò, il nostro PIO sarà, al minimo, fornito di un buffer per ogni porta.

Inoltre, abbiamo stabilito che il microcomputer userà un procedimento *handshaking*, oppure *interrupt* per comunicare con il dispositivo d'I/O. Il PIO userà anche un procedimento similare per comunicare con il periferico. Perciò, ogni PIO deve essere fornito di almeno *due linee di controllo per porta* per compiere la funzione di handshaking.

Il microprocessore avrà anche bisogno di essere in grado di leggere lo stato di ogni porta. Ogni porta deve essere fornita di uno o più *bit di stato*. Infine, esisterà un numero di opzioni all'interno di ogni PIO per configurare le sue possibilità. Il programmatore deve essere capace di accedere ad un registro speciale all'interno del PIO per specificare le opzioni di programmazione. Questo è il *registro di controllo*. In alcuni casi le informazioni di stato sono parte del registro di controllo.

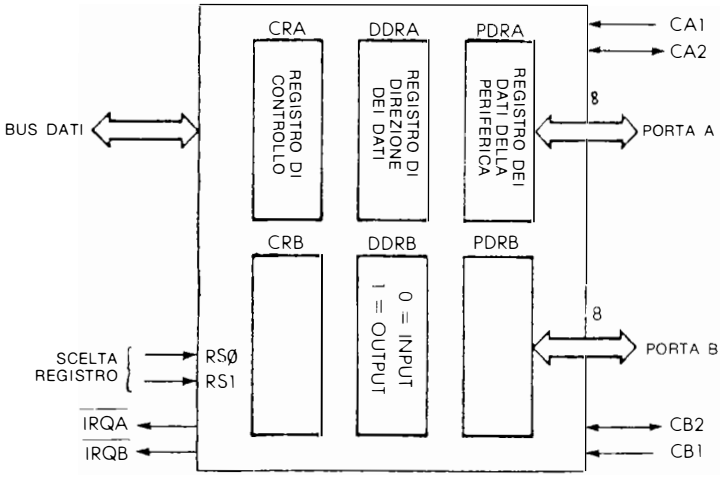


Figura 7-1: PIO Tipico

Una facoltà essenziale del PIO è il fatto che ogni linea può essere configurata come una linea d'input o come una d'output. Nell'illustrazione 7-1 appare lo schema di un PIO. Il programmatore può specificare se una linea sarà d'input o d'output. Per programmare la direzione delle linee, è fornito un *registro per la direzione dei dati* per ogni porta. Uno "0" in una posizione del bit del registro per la direzione dei dati specifica un input. Un "1" specifica un output.

Può sembrare sorprendente vedere che uno "0" sia usato per l'input e un "1" per l'output quando in realtà "0" dovrebbe corrispondere all'output e "1" all'input. Questo è intenzionale: ogni qual volta è applicata l'alimentazione al sistema, è di grande importanza che tutte le linee I/O siano configurate come *input*. In caso contrario, se il microcomputer è collegato a qualche periferico pericoloso, potrebbe attivarlo accidentalmente.

Quando è applicato un reset, tutti i registri sono normalmente azzerati e questo farà sì che tutte le linee d'input del PIO si configurino come input. Il collegamento al microprocessore appare sulla sinistra dell'illustrazione. Il PIO si collega naturalmente al bus dei dati ad 8 bit, al bus degli indirizzi del microprocessore e al bus di controllo del microprocessore. Il programmatore specificherà semplicemente l'indirizzo di qualsiasi registro a cui desidera accedere all'interno del PIO.

Il registro di controllo interno

Il Registro di Controllo del PIO fornisce un numero di opzioni per generare o percepire gli interrupt, o per compiere funzioni di handshaking automatiche. Qui non è necessaria la descrizione completa dei servizi forniti. Semplicemente, colui che usa un qualche sistema pratico che si serve di un PIO dovrà riferirsi al data-sheet (specifiche del costruttore) che mostra l'effetto dell'impostazione dei vari bit del registro di controllo. Ogni qual volta il sistema è inizializzato, il programmatore dovrà caricare il registro di controllo del PIO con il contenuto corretto per l'applicazione considerata (per una descrizione dettagliata vedere il riferimento D 380).

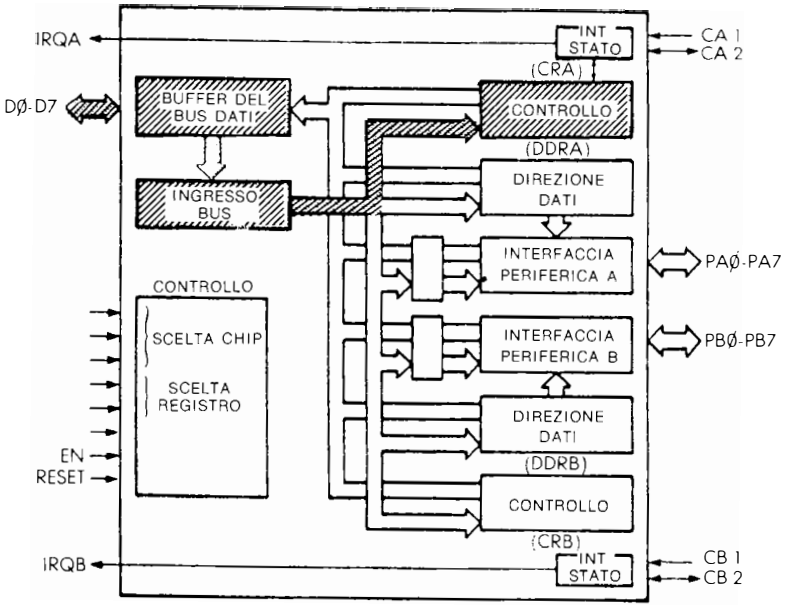


Figura 7-2: Uso del PIO - Caricamento del registro di controllo

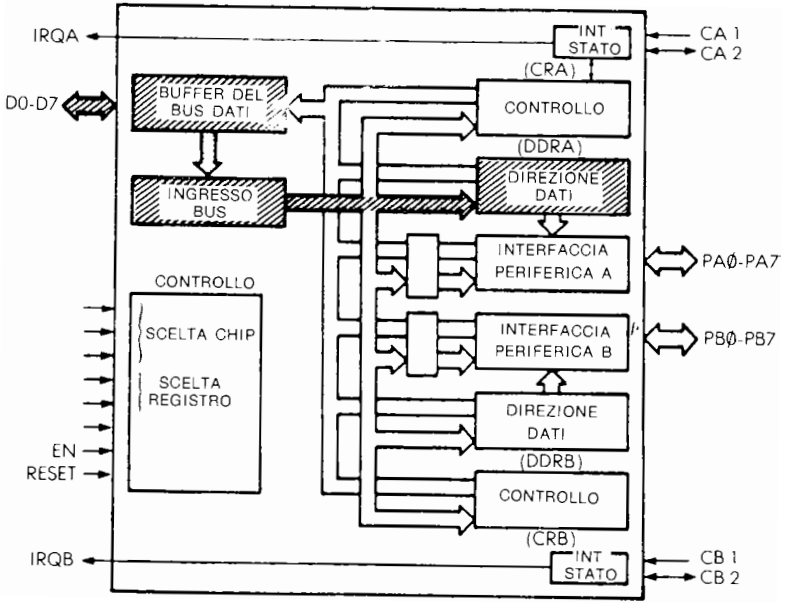


Figura 7-3: Uso del PIO - Caricamento della direzione dei dati

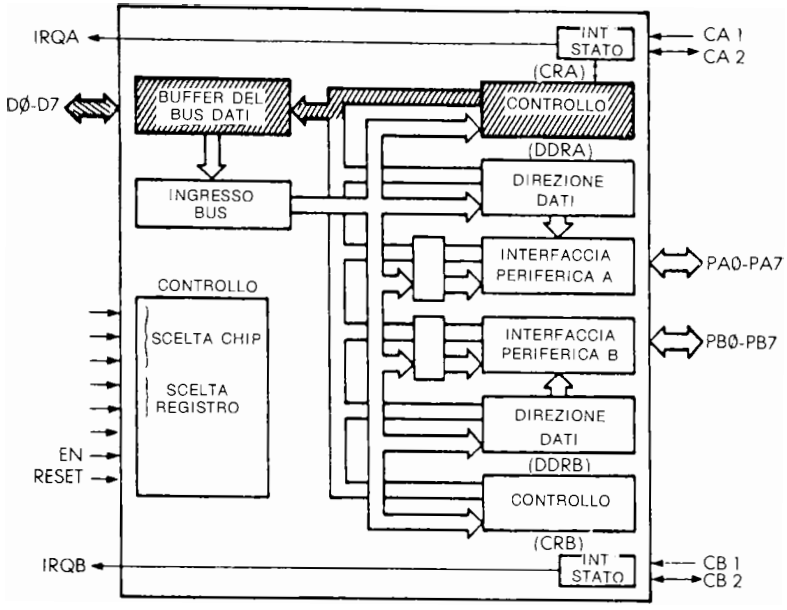


Figura 7-4: Uso del PIO - Lettura dello Stato

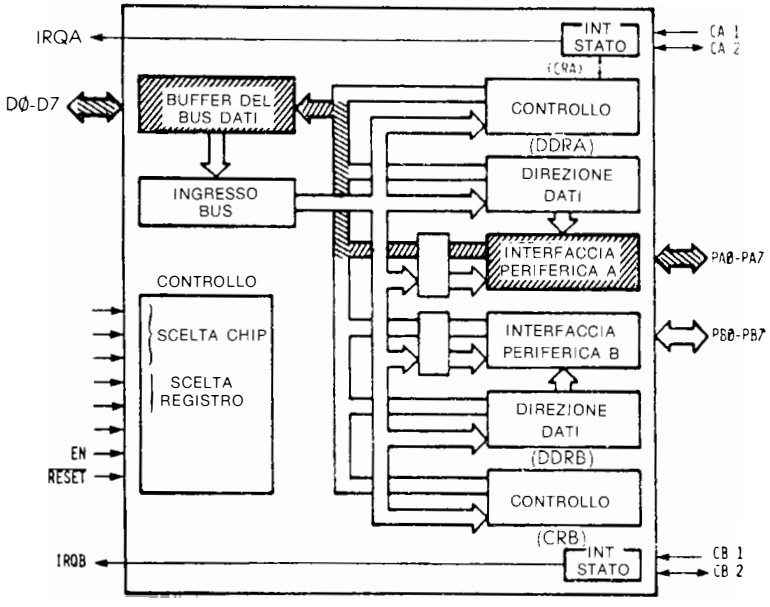


Figura 7-5: Uso del PIO - Lettura dei dati in Input

La programmazione di un PIO

Quando si usa un canale PIO, una sequenza tipica è la seguente (supponendo un input):

Caricare il registro di controllo
Ciò è eseguito tramite un trasferimento programmato tra un registro dello Z80 (di solito l'accumulatore) ed il registro di controllo del PIO. Questo regola le opzioni e il modo di funzionamento del PIO (vedere la Figura 7-2). È normalmente fatto una volta sola all'inizio di un programma.

Caricare il registro di direzione
Questo specifica la direzione in cui saranno usate le linee I O (vedere la Figura 7-3).

Leggere lo stato
Il registro di stato indica se sull'input è disponibile un byte valido. (vedere la Figura 7-4).

Leggere la porta
Il byte è letto nello Z80. (Vedere la Figura 7-5).

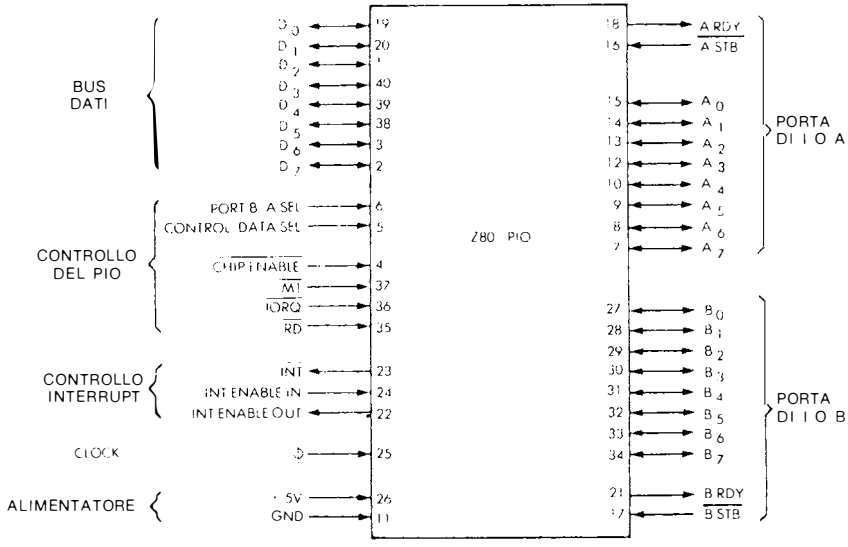


Figura 7-6: Disposizione dei pin dello Z80 PIO

Lo Z80 PIO della Zilog

Il Z80 PIO è un PIO a due porte la cui architettura è essenzialmente compatibile con il modello standard che abbiamo descritto. La configurazione reale dei pin è mostrata nella Figura 7-6, mentre nella Figura 7-7 è mostrato uno schema a blocchi.

Ogni porta del PIO ha sei registri: un registro d'input ad 8 bit, un registro d'output a 9 bit, un registro di controllo di modo a 2 bit, un registro di maschera ad 8 bit, una selezione di input/output ad otto bit (registro di direzione) ed un registro di controllo di maschera a 2 bit.

Gli ultimi tre registri sono usati soltanto quando la porta è programmata per operare nel "bit mode". Il PIO può funzionare in uno dei quattro modi, come scelto dal contenuto dei registri di controllo di modo (2 bit). Sono: output di byte, input di byte, bus bidirezionali di byte, e "bit mode".

I due bit dei registri di controllo della maschera sono caricati dal programmatore e spe-

cificano lo stato alto e basso di un dispositivo periferico che deve essere controllato, e le condizioni per cui può essere generato un interrupt.

Il registro di selezione di input/output ad 8 bit permette a qualsiasi pin di essere un input oppure un output quando opera in quel modo.

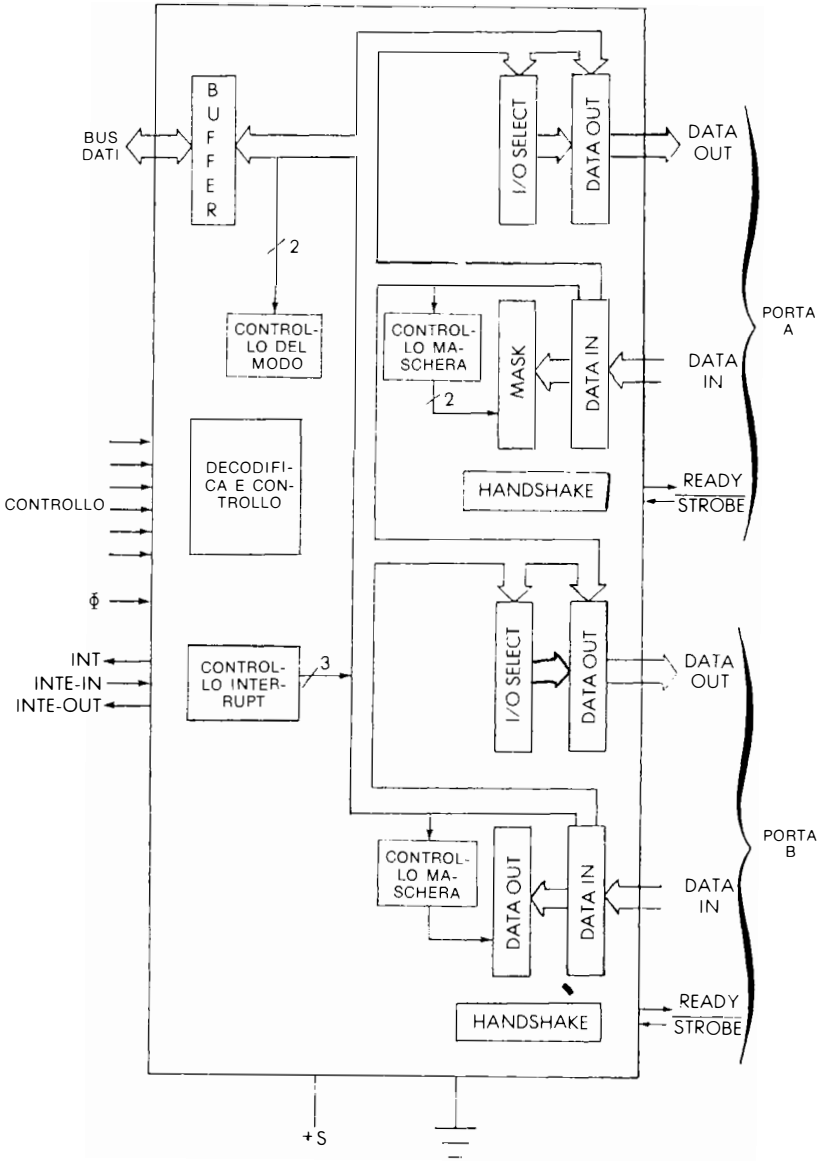


Figura 7-7: Schema a blocchi dello Z80 PIO

La programmazione del PIO Zilog

Una sequenza tipica per l'uso di un PIO, diciamo nel modo di bit, sarebbe la seguente:
Caricare il registro di controllo di modo per specificare il modo di bit.
Caricare il registro di selezione input/output della porta A per specificare che le linee 0-5 sono input e che le linee 6 e 7 sono output.
Quindi sarebbe letta una parola leggendo il contenuto del buffer d'input.
Addizionalmente, potrebbe essere usato il registro di maschera per specificare le condizioni di stato.

Lo Z80 SIO

Il SIO (Serial Input/Output) è un chip periferico a canale doppio progettato per facilitare le comunicazioni asincrone nella forma seriale. Include un UART, cioè, un trasmettitore ricevitore asincrono universale.
La sua funzione essenziale è una conversione serie-parallelo e parallelo-serie. Comunque, questo chip è fornito di possibilità sofisticate, come la manipolazione automatica di complessi protocolli orientati al byte (byte-oriented protocols) come l'IBM bisync cosi come HDLC e SDLC.
Addizionalmente, può funzionare in modo sincrono come un USRT e generare e controllare i codici CRC. Offre una scelta di chiamate, interrupt e di modi per il trasferimento dei blocchi.

Altri chip di I/O

Siccome lo Z80 è comunemente usato al posto dello 8080, è stato progettato in modo che possa essere associato con quasi tutti i soliti chip d'input/output dell'8080, cosi come i chip I/O specifici prodotti da Zilog. Tutti i chip d'input/output dell'8080 possono essere considerati per l'uso di un sistema Z80.

SOMMARIO

Per fare un uso efficace dei componenti d'input/output è necessario comprendere dettagliatamente la funzione di ogni bit, o gruppo di bit all'interno dei vari registri di controllo. Questi nuovi chip complessi automatizzano un numero di procedure che dovevano essere compiute prima via software o dalla logica speciale. In particolare, un buon numero delle procedure di handshoking sono automatizzate all'interno di componenti come il SIO. Può essere interna anche la manipolazione e la rivelazione degli interrupt. Con le informazioni che sono state presentate nel precedente capitolo, il lettore dovrebbe essere in grado di capire quali sono le funzioni dei segnali fondamentali e dei registri. Naturalmente, saranno introdotti componenti più nuovi i quali offriranno una esecuzione hardware di algoritmi ancor più complessi.

CAPITOLO 8

ESEMPI APPLICATIVI

INTRODUZIONE

Questo capitolo è progettato per provare le vostre nuove capacità di programmazione presentando una raccoltà di programmi utili. Questi programmi o "routine" sono frequentemente incontrati in applicazioni, e sono generalmente chiamate "utility routines". Richiederanno una sintesi della conoscenza e delle tecniche presentate fino a questo punto.

Preleveremo caratteri da un dispositivo d'I/O e li elaboreremo in vari modi. Ma prima di tutto, cancelliamo un'area della memoria (questo può non essere necessario — ognuno di questi programmi è presentato soltanto come un esempio di programmazione).

CANCELLAZIONE DI UNA SEZIONE DELLA MEMORIA

Vogliamo cancellare il contenuto della memoria dall'indirizzo BASE + 1 all'indirizzo BASE + LENGTH, dove LENGTH è minore di 256.

Il programma è:

```
ZEROM   LD   B, LENGTH    CARICA B CON LENGTH
        LD   A,0          CANCELLA A
        LD   HL, BASE     PUNTA ALLA BASE
CLEAR   LD   (HL), A      CANCELLA UNA LOCAZIONE
        INC  HL           PUNTA ALLA PROSSIMA
        DEC  B            DECREMENTA IL CONTATORE
        JR   NZ, CLEAR    FINE DELLA SEZIONE?
        RET
```

Nel programma precedente la lunghezza della sezione della memoria è supposta sia uguale a LENGTH. La coppia di registri HL è usata come puntatore alla parola corrente che sarà cancellata. Come al solito, il registro B è usato come contatore.

L'accumulatore A è caricato solo una volta con il valore 0 (tutti zero), poi è copiato nelle locazioni di memoria successive.

In un programma di prova della memoria, per esempio, questa routine di utilità potrebbe essere usata per azzerare il contenuto di un blocco. Poi il programma di prova della memoria dovrebbe di solito verificare che il suo contenuto rimanga 0.

Quella precedente era una facile realizzazione di una routine di cancellazione. Miglioriamola.

Il programma migliorato compare sotto:

```
ZEROM   LD   B, LENGTH
        LD   HL, BASE
LOOP    LD   (HL), 0
        INC  HL
        DJNZ LOOP
        RET
```

I due miglioramenti sono stati ottenuti eliminando l'istruzione LDA, 0, caricando uno "zero" direttamente nella posizione puntata da H ed L ed usando anche l'istruzione DJNZ speciale dello Z80.

Questo esempio di miglioramento dovrebbe dimostrare che *ogni volta che è scritto un programma, anche se può essere giusto, di solito può essere migliorato esaminandolo attentamente*. Per eseguire tali miglioramenti è essenziale una dimestichezza con il set completo delle istruzioni. Questi miglioramenti non sono solo cosmetici. Migliorano il tempo di esecuzione del programma, richiedono meno istruzioni e perciò meno spazio di memoria, e generalmente migliorano anche la leggibilità del programma e, perciò, le sue possibilità di essere corretto.

Esercizio 8-1: *Scrivete un programma di prova della memoria che azzeri un blocco di 256 parole, e che poi verifichi che ogni posizione sia 0. In seguito dovrà scrivere tutti 1 e verificherà il contenuto del blocco. Poi scriverà 01010101, e verificherà il contenuto. Alla fine, scriverà 10101010, e verificherà il contenuto.*

Esercizio 8-2: *Modificate il programma di sopra in modo che riempia la sezione di memoria con 0 e 1 alternati (tutti 0, poi tutti 1).*

Facciamo un polling sui nostri dispositivi I/O per trovare quale richiede servizio.

POLLING DEI DISPOSITIVI DI I/O

Supporremo che questi dispositivi di I/O siano collegati al nostro sistema. I loro registri di stato sono disposti agli indirizzi IOSTATUS1, IOSTATUS2, IOSTATUS3. Il programma è:

```
TEST    IN   A, (STATUS1)  LEGGI IOSTATUS1
        BIT  7,A           PROVA IL BIT "READY" (BIT 7)
        JP   NZ, FOUND1    SALTA AL MANIPOLATORE 1
        IN   A, (STATUS2)  LO STESSO PER IL DISPOSITIVO 2
        BIT  7,A
        JP   NZ, FOUND2
        IN   A, (STATUS3)  LO STESSO PER IL DISPOSITIVO 3
        BIT  7,A
        JP   NZ, FOUND3
```

La "MASK" conterrà, per esempio, "10000000" se proviamo la posizione di bit 7. Come risultato dell'istruzione BIT, il bit Z dei flag di stato sarà posto ad 1 se "MASK AND STATUS" non è zero, cioé, se il bit corrispondente di STATUS concorda con quello di MASK. L'istruzione JP NZ (salta se non è uguale a zero) porterà ad una ramificazione alla routine FOUND appropriata.

ACCETTAZIONE DI CARATTERI D'INGRESSO

Supponiamo di aver appena trovato che un carattere è pronto sulla tastiera. Accumuliamo caratteri nell'area di memoria chiamata BUFFER fino a quando incontriamo un carattere speciale chiamato SPC, il cui codice è stato definito precedentemente.

La subroutine GETCHAR preleverà un carattere dalla tastiera (vedere il Capitolo 6 per ulteriori dettagli) e lo lascierà nell'accumulatore. Supponiamo che 256 caratteri al massimo siano prelevati prima che sia trovato un carattere SPC.

```
STRING   LD   HL, BUFFER   PUNTA AL BUFFER
NEXT     CALL GETCHAR      PRENDI UN CARATTERE
         CP   SPC          CONTROLLA SE È PRESENTE
                           IL CARATTERE SPECIALE
         JR   Z, OUT       TROVATO?
         LD   (HL), A      IMMAGAZZINA CARATTERE IN BUFFER
         INC  HL           PROSSIMA POSIZIONE BUFFER
         JR   NEXT         PRENDI IL PROSSIMO CARATTERE
OUT      RET
```

Esercizio 8-3: *Miglioriamo questa routine fondamentale:*
a Fate ritornare l'eco del carattere al dispositivo (per una Telescrivente, per esempio)
b — controllate che la stringa d'input non sia più lunga di 256 caratteri.

Ora abbiamo una stringa di caratteri in un buffer di memoria. Elaboriamoli in vari modi.

PROVA DI UN CARATTERE

Determiniamo se il carattere alla posizione di memoria LOC è uguale a 0, 1 oppure 2:

```
ZOT      LD   A, (CHAR)    ACCETTA CARATTERE
         CP   00           È UNO ZERO?
         JP   Z, ZERO      SALTA ALLA ROUTINE
         CP   01           UN UNO?
         JP   Z, ONE
         CP   02           UN DUE?
         JP   Z, TWO
         JP   NOTFND       GUASTO
```

Noi leggiamo semplicemente il carattere, poi usiamo l'istruzione CP per controllare il suo valore.

Ora eseguiamo una prova diversa.

PROVA PARENTESI

Determiniamo se il carattere ASCII alla posizione di memoria LOC è un digit tra 0 e 9:

```
BRACK    LD   A, (CHAR)    ACCETTA CARATTERE
         AND  7FH          MASCHERA IL BIT DI PARITÀ
         CP   30H          ASCII 0
         JR   C, OUT       CARATTERE TROPPO BASSO?
         CP   39H          ASCII 9
         JR   NC, OUT      CARATTERE TROPPO ALTO?
         CP   A            FORZA IL FLAG ZERO
OUT      RET  EXIT
```

L'ASCII "0" è rappresentato in esadecimale da "30" o da "D0" a seconda se il bit di parità è usato o no. Allo stesso modo, l'ASCII "9" è rappresentato in esadecimale da "39" o da "B9".

Lo scopo della seconda istruzione del programma è quello di cancellare il bit 7, il bit di pa-

rità, nel caso considerato, in modo che il programma è applicabile ad entrambi i casi. Il valore del carattere è poi paragonato ai valori ASCII per "0" e "9".

Quando si usa una istruzione di confronto, il flag Z è posto ad uno se riesce il confronto. Il bit di riporto è posto ad uno in caso di prestito, altrimenti è posto a 0. In altre parole, quando si usa l'istruzione CP, il bit di riporto sarà posto a uno se il valore del literal che compare nell'istruzione è più grande del valore contenuto nell'accumulatore. Sarà posto a zero se è minore o uguale.

L'ultima istruzione, CPA, forza uno "0" nel flag Z. Il flag Z è usato per indicare alla routine chiamante che il carattere in CHAR era davvero nell'intervallo (0,9). Possono essere usate altre convenzioni, come quella di caricare un digit nell'accumulatore per indicare il risultato della prova.

Esercizio 8-4: *Il seguente programma è equivalente a quello precedente?:*

```
LD    A, (CHAR)
SUB   30H
JP    M, OUT
SUB   10
JP    P, OUT
ADD   10
```

Esercizio 8-5: *Determinate se un carattere ASCII contenuto nell'accumulatore è una lettera dell'alfabeto.*

Quando usate una tabella ASCII, noterete che è spesso usata la parità. Per esempio, l'ASCII per "0" è "0110000", un codice da 7 bit. Comunque, se per esempio usiamo la parità dispari, noi garantiamo che il numero totale degli uni in una parola è dispari; allora il codice diventa: "10110000". È aggiunto un "1" extra a sinistra. Questo è "B0" in esadecimale. Perciò, sviluppiamo un programma per generare la parità.

GENERAZIONE DELLA PARITÀ

Questo programma genererà una parità pari nella posizione di bit 7:

```
PARITY   LD    A, (CHAR)    ACCETTA CARATTERE
         AND   7 FH         CANCELLA IL BIT DI PARITÀ
         JR    PE, OUT      CONTROLLA SE LA PARITA È GIÀ PARI
         OR    80H          PONI AD 1 IL BIT DI PARITÀ
OUT      LD    (LOC), A     IMMAGAZZINA IL RISULTATO
```

Il programma usa il circuito di rivelazione della parità interna disponibile nello Z80.

La terza istruzione: JR PE, OUT controlla se la parità della parola nell'accumulatore è già pari. Questa prova sarà soddisfatta se la parità è pari, "PE", e uscirà.

Se la parità non è pari, cioé, se l'istruzione di salto non è soddisfatta, allora la parità è dispari e deve essere scritto un "1" nella posizione di bit 7.

Questo è lo scopo della quarta istruzione:

```
OR   80H
```

Alla fine, il valore che ne risulta è salvato nella posizione di memoria LOC.

Esercizio 8-6: *Il problema precedente era troppo semplice da risolvere, usando il circuito interno di rivelazione della parità. Come esercizio, vi è richiesto di risolvere lo stesso problema senza usare questo circuito. Spostate il contenuto dell'accumulatore, e contate il numero degli 1 per determinare quale bit dovrebbe essere scritto nella posizione di parità.*

Esercizio 8-7: *Verificate la parità di una parola usando il programma precedente come esempio. Dovete calcolare la parità corretta, poi confrontarla a quella supposta.*

CONVERSIONE DI CODICE: DA ASCII A BCD

La conversione da ASCII a BCD è molto semplice.

Osserveremo che la rappresentazione esadecimale dei caratteri ASCII da 0 a 9 è da 30 a 39 oppure da B0 a B9, a secondo della parità. La rappresentazione BCD è ottenuta semplicemente facendo cadere il "3" o il "B", cioé, mascherando il nibble sinistro (4 bit):

```
ASCBCD    CALL  BRACK        CONTROLLA CHE IL CARATTERE SIA DA 0 A 9
          JP    NZ, ILLEGAL  USCITA SE IL CARATTERE NON È CONSENTITO
          LD    A, (CHAR)    PRENDI IL CARATTERE
          AND   OF           CANCELLA IL NIBBLE ALTO
          LD    (BCDCHAR), A IMMAGAZZINA IL RISULTATO
```

Esercizio 8-8: *Scrivere un programma per convertire il BCD in ASCII.*

Esercizio 8-9: *Scrivere un programma per convertire il BCD in binario (più difficile).*

Traccia: $N_3\ N_2\ N_1\ N_0$ nel BCD è $(((N_3 \times 10) + N_2) \times 10 + N_1) \times 10 + N_0$ nel binario.

Per moltiplicare per 10, usare uno spostamento a sinistra ($= \times 2$), un altro spostamento a sinistra ($\times 4$), un ADC ($= \times 5$), un altro spostamento a sinistra ($\times 10$).

Nella notazione BCD completa, la prima parola può contenere il conteggio dei digit BCD, il nibble successivo può contenere il segno, ed ogni altro nibble può contenere un digit BCD (supponiamo nessun punto decimale). L'ultimo nibble del blocco può non essere usato.

CONVERSIONE DA ESADECIMALE IN ASCII

"A" contiene un digit esadecimale. Abbiamo bisogno semplicemente di sommare un "3" (o un "B") nel nibble sinistro:

```
          AND   FH         AZZERA IL NIBBLE SINISTRO (opzionale)
          ADD   A, 30H     ASCII
          CP    A, 3AH     CORREZIONE NECESSARIA?
          JP    M, OUT
          ADD   A, 7       CORREZIONE PER I CARATTERI DA A AD F
```

Esercizio 8-10: *Convertire il codice esadecimale in ASCII, supponendo un formato compatto (due digit esadecimali in A).*

RICERCA DELL'ELEMENTO PIÙ GRANDE DI UNA TABELLA

L'indirizzo d'inizio della tabella è contenuto all'indirizzo di memoria BASE nella pagina zero. La prima entrata della tabella è il numero di byte che esso contiene. Questo programma ricercherà l'elemento più grande della tabella. Il suo valore sarà lasciato in A, e la sua posizione sarà immagazzinata nella posizione di memoria INDEX.

Questo programma usa i registri A, B, H, L ed userà l'indirizzamento indiretto, in modo che possa cercare una tabella dovunque nella memoria (vedere la Figura 8-1).

```
MAX       LD    HL, BASE     INDIRIZZO DELLA TABELLA
          LD    B, (HL)      NUMERO DI BYTE NELLA TABELLA
          LD    A, 0         AZZERA IL VALORE MASSIMO
```

```
        LD    (INDEX), HL    INIZIALIZZA L'INDICE
        INC   HL             PROSSIMA ENTRATA
LOOP    CP    (HL)           PARAGONA L'ENTRATA
        JR    NC,NOSWITCH    SALTA SE È INFERIORE ALLA MASSIMA
        LD    A, (HL)        CARICA IL NUOVO VALORE MASSIMO
        LD    (INDEX) HL     CARICA IL NUOVO VALORE MASSIMO
NOSWITCH INC  HL             PUNTA ALLA PROSSIMA ENTRATA
        DEC   B              DECREMENTA IL CONTATORE
        JR    NZ, LOOP       CONTINUA SE NON È ZERO
        RET
```

Questo programma prova prima la ennesima entrata. Se è maggiore di 0, l'entrata va in A e la sua posizione è ricordata nell'INDEX. Poi è provata l'entrata (n−1)−esima e cosi via. Questo programma funziona per numeri interi positivi.

Esercizio 8-11: *Modificate il programma in modo che funzioni anche per numeri negativi in complemento a due.*

Esercizio 8-12: *Questo programma lavorerà anche per i caratteri ASCII?*

Esercizio 8-13: *Scrivete un programma che classifichi n numeri in ordine crescente.*

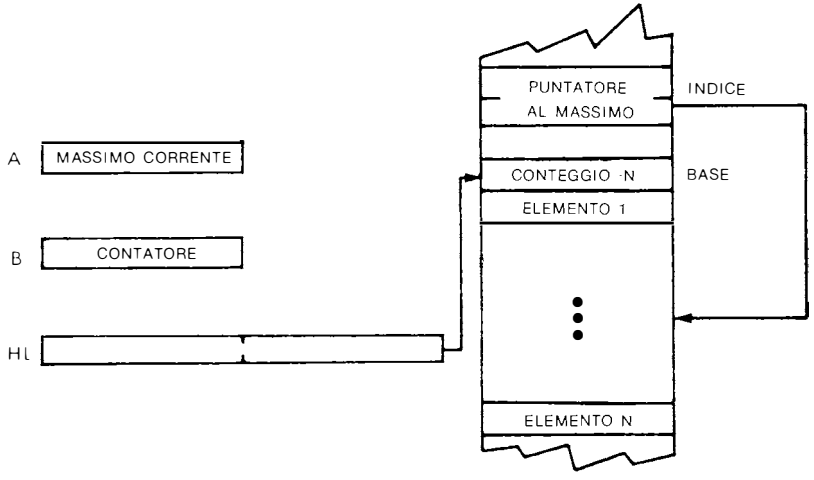


Figura 8-1: L'elemento più grande di una tabella

Esercizio 8-14: *Scrivete un programma che classifichi n nomi (3 caratteri ognuno) in ordine alfabetico.*

SOMMA DI N ELEMENTI

Questo programma calcolerà la somma a 16 bit delle N entrate di una tabella. L'indirizzo d'inizio della tabella è contenuto all'indirizzo di memoria BASE, nella pagina ZERO. La prima entrata della tabella contiene il numero N degli elementi. La somma a 16 bit sarà lasciata nelle posizioni di memoria SUMLO e SUMHI. Se la somma dovesse richiedere più di 16 bit, saranno tenuti solo i 16 più bassi. (Si dice che i bit di ordine alto sono troncati).

Questo programma modificherà i registri A, B, H, L ed IX.
Si suppone un massimo di 256 elementi (vedere la Figura 8-2).

```
SUMIG    LD    HL, BASE        PUNTA ALLA BASE DELLA TABELLA
         LD    B, (HL)         LEGGI LA LUNGHEZZA NEL CONTATORE
         INC   HL              PUNTA ALLA PRIMA ENTRATA
         LD    IX, SUMLO       PUNTA AL RISULTATO, BASSO
         LD    A, 0            AZZERA IL RISULTATO
         LD    (IX + 0), A     BASSO
         LD    (IX + 1), A     ED ALTO
ADLOOP   LD    A, (HL)         ACCETTA L'ENTRATA DELLA TABELLA
         ADD   A, (IX + 0)     CALCOLA LA SOMMA PARZIALE
         LD    (IX + 0), A     IMMAGAZZINALA
         JR    NC, NOCARRY     CONTROLLA SE C'È RIPORTO
         INC   (IX + 1)        SOMMA IL RIPORTO AL BYTE ALTO
NOCARRY  INC   HL              PUNTA ALLA PROSSIMA ENTRATA
         DEC   B               DECREMENTA IL CONTEGGIO DI BYTE
         JR    NZ, ADLOOP      CONTINUA A SOMMARE FINO ALLA FINE
         RET
```

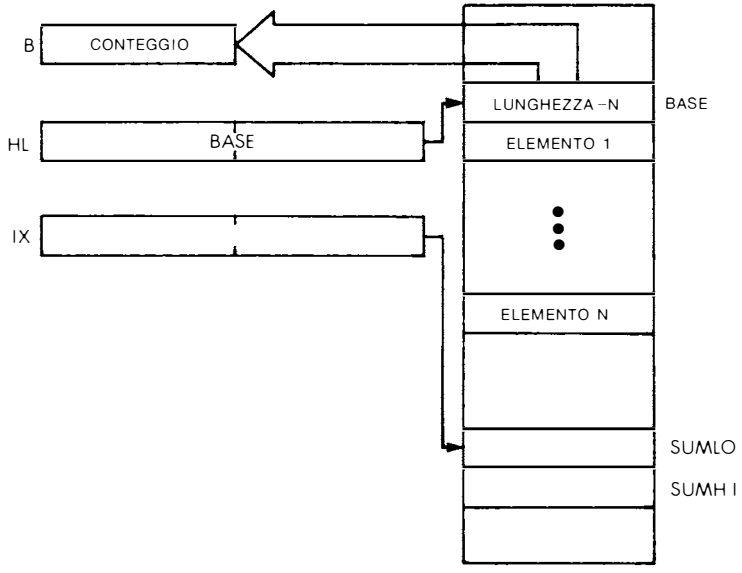


Figura 8-2: Somma di N elementi

Questo programma è facile e dovrebbe essere autoesplicativo.

Esercizio 8-15: *Modificate questo programma per:*
a– calcolare una somma a 24 bit
b– calcolare una somma a 32 bit
c– rivelare qualsiasi eccedenza di capacità (overflow).

UN CALCOLO DI CHECKSUM

Un checksum è un digit o una serie di digit prelevati da un blocco di caratteri successivi. Il checksum è calcolato nel momento in cui il dato è immagazzinato e poi messo alla fine. Per verificare l'integrità del dato, il dato è letto, poi checksum è ricalcolato e paragonato al valore immagazzinato. Una differenza indica un errore o un guasto.

Sono usati diversi algoritmi. Qui, faremo l'OR esclusivo di tutti i byte in una tabella di N elementi e lasceremo il risultato nell'accumulatore. Come al solito, la base della tabella è immagazzinata all'indirizzo BASE nella pagina zero. La prima entrata della tabella è il suo numero di elementi N. Il programma modifica A, B, H ed L. N deve essere inferiore a 256.

```
CHECKSUM  LD    HL,BASE        INDIRIZZO DI GUIDA DELLA TABELLA IN HL
          LD    B,(HL)         ACCETTA N = LUNGHEZZA
          XOR   A              CANCELLA IL CHECKSUM
          INC   HL             PUNTA AL PRIMO ELEMENTO
CHLOOP    XOR   (HL)           CANCELLA IL CHECKSUM
          INC   HL             PUNTA AL PROSSIMO ELEMENTO
          DEC   B              DECREMENTA IL CONTATORE
          JR    NZ,CHLOOP      ESEGUI DI NUOVO SE NON FINITO
          LD    (CHECKSUM),A   CONSERVA IL CHECKSUM
          RET
```

CONTEGGIO DI ZERI

Questo programma conterà il numero di zeri nella nostra solita tabella, e lo caricherà nella posizione TOTAL. Esso modifica A, B, C, H ed L.

```
ZEROS     LD    HL,BASE        PUNTA ALLA TABELLA
          LD    B,(HL)         LEGGI LA LUNGHEZZA NEL CONTATORE
          LD    C,0            TOTALE DEGLI ZERI
          INC   HL             PUNTA ALLA PRIMA ENTRATA
ZLOOP     LD    A,(HL)         ACCETTA ELEMENTO
          OR    0              PONI AD 1 IL FLAG ZERO
          JR    NZ,NOTZ        È UNO ZERO?
          INC   C              SE SÌ, INCREMENTA IL CONTEGGIO DEGLI ZERI
NOTZ      INC   HL             PUNTA ALLA PROSSIMA ENTRATA
          DEC   B              DECREMENTA IL CONTATORE DELLA LUNGHEZZA
          JR    NZ,ZLOOP
          LD    A,C
          LD    (TOTAL),A      SALVA IL CONTEGGIO DI ZERI
```

Esercizio 8-16: *Modificate questo programma per contare*
a — il numero di asterischi ("")*
b — il numero di lettere dell'alfabeto
c — il numero di digit tra "0" e "9"

TRASFERIMENTO DI BLOCCHI

Prendiamo ogni terza entrata nel blocco di sorgente all'indirizzo FROM e immagazziniamolo in un blocco all'indirizzo TO:

```
FER 3     LD    HL,FROM
          LD    DE,TO          POSIZIONA I PUNTATORI
          LD    BC,SIZE
```

```
LOOP    LDI                 TRASFERIMENTO AUTOMATICO
        INC  HL
        INC  HL             SALTA 2 ENTRATE
        JR   PE, LOOP
```

TRASFERIMENTO DI BLOCCHI BCD

Spingeremo in alto i digit BCD nella memoria, cioè, sposteremo i nibble di 4 bit (vedere la Figura 8-3). Il programma compare sotto:

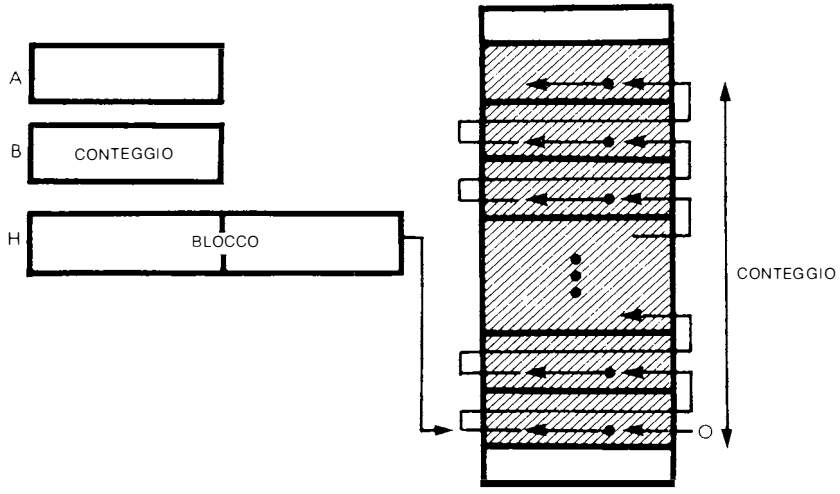


Figura 8-3: Trasferimento di blocchi BCD — La memoria

```
DMOV    LD   B, COUNT
        LD   HL, BLOCK
        XOR  A              A = 0
LOOP    RLD
        INC  HL             PUNTA AL PROSSIMO BYTE
        DJNZ LOOP           DECREMENTA IL LOOP DI CONTEGGIO FINO A ZERO
```

Il programma usa l'istruzione RLD, che non abbiamo ancora usato.
RLD ruota un digit BCD a sinistra tra A e (HL).
(HL) oppure M designano il contenuto della posizione di memoria puntata tramite H e L.

```
M LOW  va nell'M HIGH
M HIGH va nell'A LOW
A LOW  va nell'M LOW
```

Qui, "low" (basso) e "high (alto) si riferiscono ad un bibble di 4 bit.
Per usare la potente istruzione DJNZ, il registro B è usato come contatore di digit. HL è impostato per puntare all'inizio del blocco.
A è usato per immagazzinare il digit sinistro spostato da ogni rotazione tra due accessi successivi al blocco.
Per convenzione, "0" sarà fatto entrare nella parte bassa del blocco.

CONFRONTO DI DUE NUMERI CON SEGNO A 16 BIT

IX punta al primo numero N1.
IY punta ad N2 (vedere la Figura 8-4).
Il programma pone ad 1 il bit di riporto se N1 < N2, e il bit Z se N1 = N2.

```
COMP    LD    B, (IX+1)     ACCETTA IL SEGNO DI N1
        LD    A, B
        AND   80H           PROVA IL SEGNO, AZZERA CY
        JR    NZ, NEGM1     N1 È NEGATIVO
        BIT   7, (IY+1)
        RET   NZ            N2 È NEGATIVO
        LD    A, B
        CP    (IY+1)        I SEGNI SONO ENTRAMBI POSITIVI
        RET   NZ
        LD    A, (IX)
        CP    (IY)
        RET
NEGM1   XOR   (IY+1)
        RLA                 BIT DI SEGNO NEL CY
        RET   C             SEGNI DIVERSI
        LD    A, B
        CP    (IY+1)        ENTRAMBI SEGNI NEGATIVI
        RET   NZ
        LD    A, (IX)
        CP    (IY)
        RET
```

Il programma per prima cosa prova i segni di N1 ed N2. Se N1 è negativo, avverrà un salto a NEGM1. Altrimenti, viene eseguita la parte alta del programma.

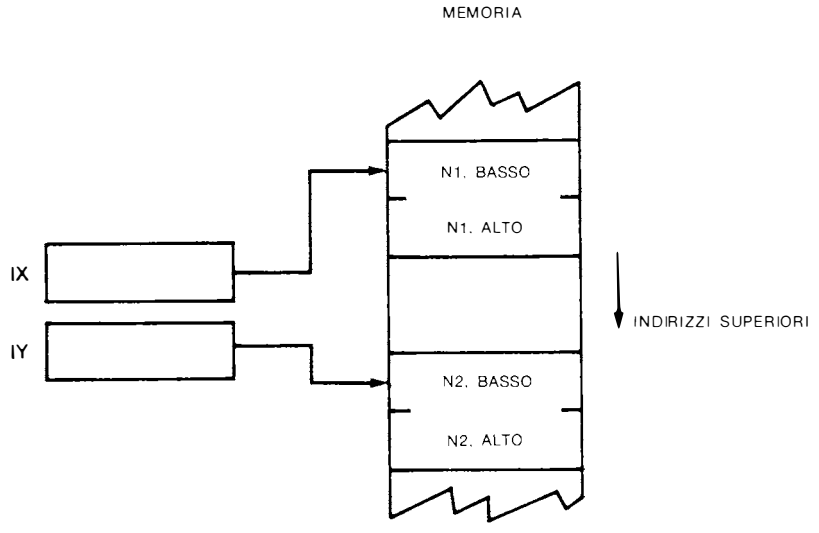


Figura 8-4: Confronto tra due numeri con segno

Notate che l'istruzione BIT è usata nella quinta linea per provare direttamente il bit di segno di N2 nella memoria:

```
        BIT    7, (IY+1)
```

Lo stesso poteva essere fatto per N1, ma d'altra parte fra breve avremo bisogno del valore di N1. È perciò più semplice leggere N1 dalla memoria e conservarlo in B:

```
COMP    LD     B, (IX+1)
```

È necessario conservare N1 in B perchè l'AND può distruggere il contenuto di A:

```
        LD     A, B
        AND    80H
```

Notate che è usato anche un ritorno condizionale (linea 6):

```
        RET    NZ
```

Questa è una caratteristica notevole dello Z80 che semplifica la programmazione.

Notate che l'istruzione di paragone opera direttamente sul contenuto della memoria, nel modo indicizzato:

```
        CP     (IY+1)
```

Quando si confrontano i due numeri, il byte più significativo è paragonato per primo e per secondo quello meno significativo.

Notate l'uso estensivo del meccanismo d'indicizzazione in questo programma, il che porta a una codifica efficiente.

BUBBLE-SORT

Il bubble-sort è una tecnica di classificazione usata per ordinare gli elementi di una tabella in ordine crescente o decrescente. La tecnica del bubble-sort deriva il suo nome dal fatto che l'elemento più piccolo "gorgoglia" (bubble-up) alla parte alta della tabella. Ogni volta che "si scontra" con un elemento "più pesante", gli salta sopra.

Nella Figura 8-5 è mostrato un esempio pratico del bubble-sort.

L'elenco che deve essere classificato contiene: (10, 5, 0, 2, 100), e deve essere classificato in ordine crescente ("0" in alto). L'algoritmo è semplice e nella Figura 8-7 è mostrato il diagramma di flusso.

Sono confrontati i due elementi più in alto (oppure i due più in basso). Sono scambiati se quello più in basso è inferiore ("più leggero") di quello più in alto. Nel caso contrario non lo sono. Per scopi pratici, lo scambio, se avviene, sarà ricordato in un flag chiamato "EXCHANGED". Il processo è poi ripetuto sulla prossima coppia di elementi, ecc., fino a che tutti gli elementi sono stati confrontati due alla volta.

Questo primo passo è illustrato dalle fasi 1, 2, 3, 4, 5, 6 nella Figura 8-5 andando dal basso in lato. (Allo stesso modo potremmo andare dall'alto in basso).

La classificazione è completa se non è stato scambiato nessun elemento. Se è avvenuto uno scambio, ricominciamo tutto da capo.

Guardando la Figura 8-6, si può vedere che in questo esempio sono necessari quattro passi.

Il procedimento è semplice ed è largamente usato.

Una complicazione addizionale risiede nel meccanismo vero dello scambio.

Quando si scambia A e B, non si può scrivere

```
A=B
B=A
```

siccome questo comporterebbe una perdita del valore precedente di A (provatelo su un esempio).

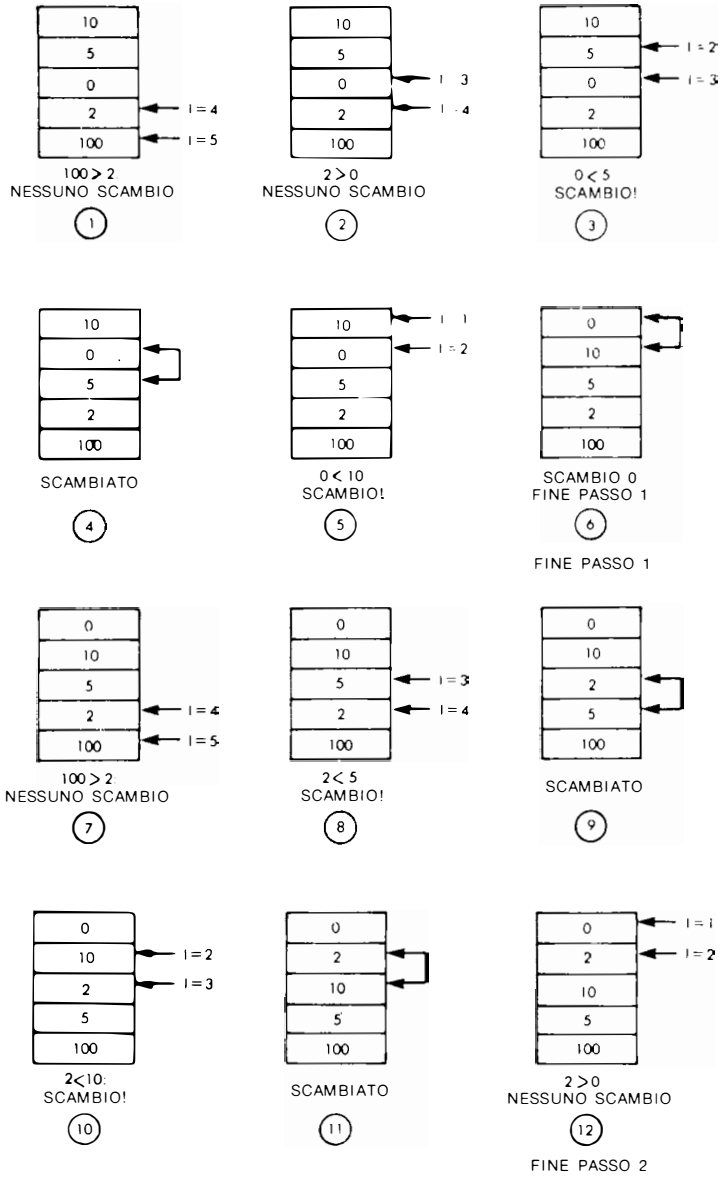


Figura 8-5: Esempio di Bubble-Sort: Fasi da 1 a 12

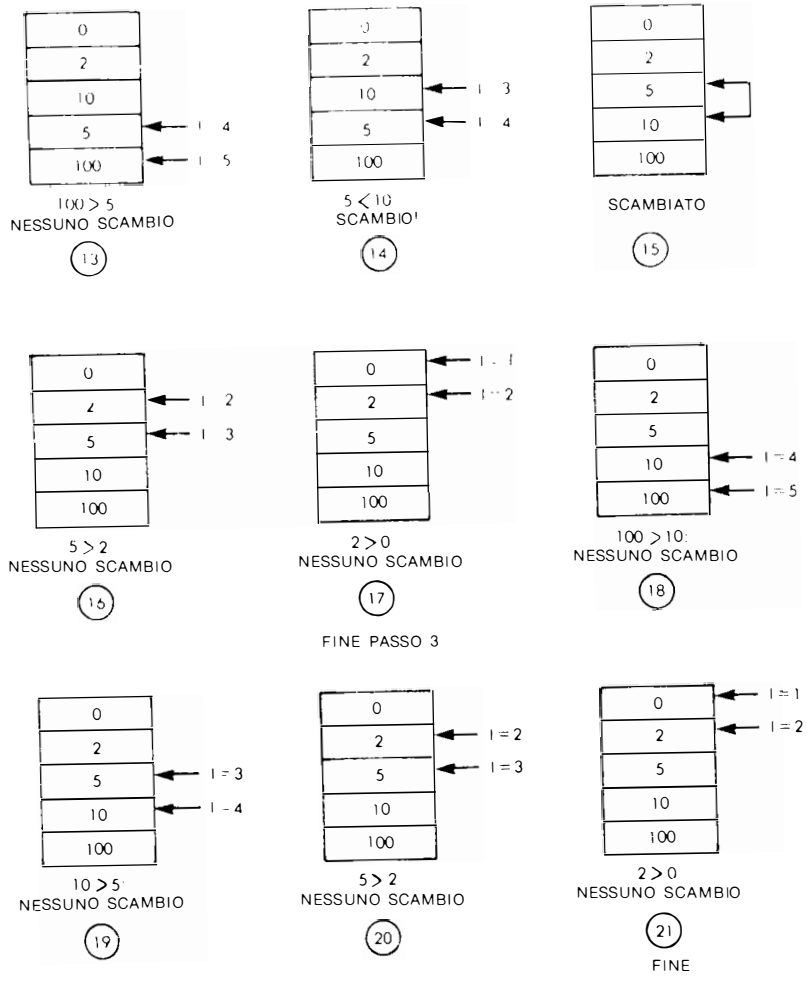


Figura 8-6: Esempio di Bubble-Sort: Fasi da 13 a 21

La soluzione corretta è di usare una variabile temporanea o posizione per conservare il valore di A:

```
TEMP = A
A    = B
B    = TEMP
```

Cosi funziona (provatelo su un esempio). Questa è chiamata permutazione circolare. Questo è il modo in cui i programmi compiono lo scambio. Questa tecnica è illustrata sul diagramma di flusso della Figura 8-7.

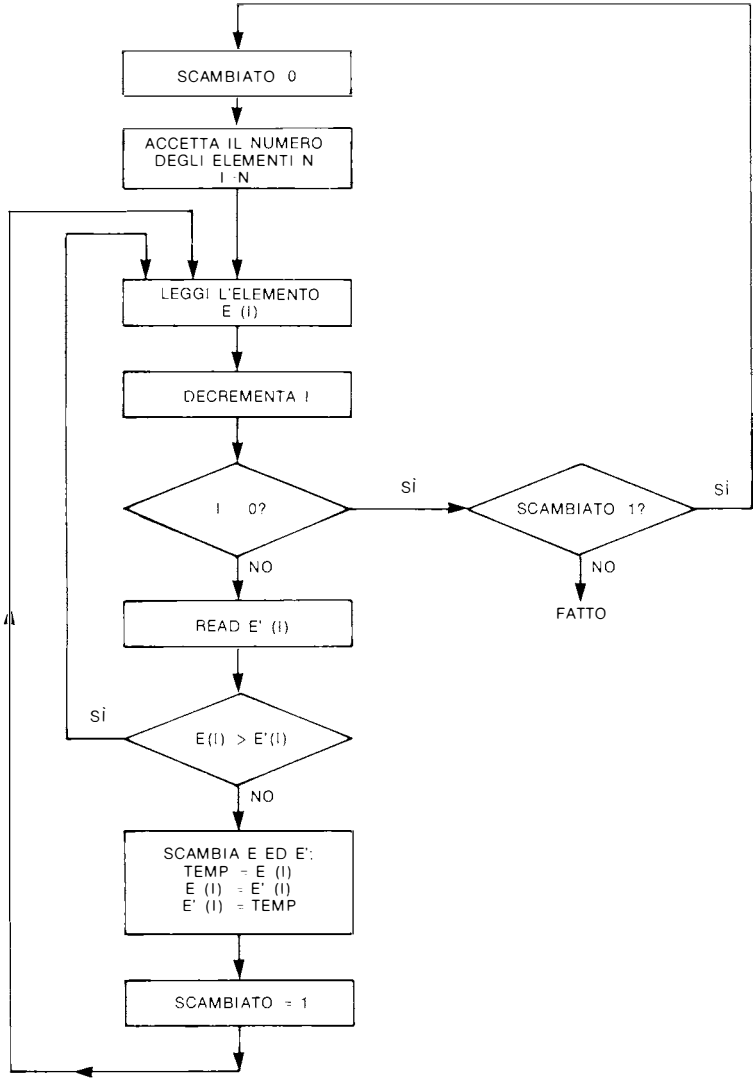


Figura 8-7: Diagramma di flusso del Bubble-Sort

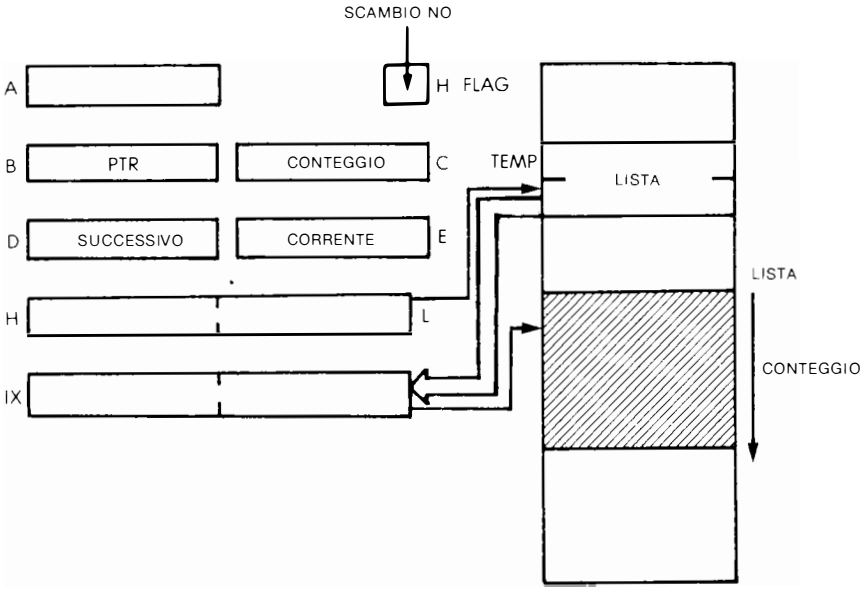


Figura 8-8: Bubble-Sort

I trasferimenti dei registri e della memoria sono mostrati nella Figura 8-8, ed il programma è:

```
BUBBLE   LD    (TEMP), HL     TEMP = (H, L)
AGAIN    LD    IX, (TEMP)     IX = (HL)
         RES   FLAG, H        FLAG SCAMBIATO = 0
         LD    B, C
         DEC   B
NEXT     LD    A, (IX)
         LD    D, A           D = ENTRATA CORRENTE
         LD    E, (IX + 1)    E = PROSSIMA ENTRATA
         SUB   E              CONFRONTO
         JR    NC, NOSWTCH    VAI A NOSWITCH
                              SE ENTRATA CORRENTE > ENTRATA
                              SUCCESSIVA
XCHANGE  LD    (IX), E        IMMAGAZZINA SUCCESSIVA
                              IN CORRENTE
         LD    (IX + 1), D    IMMAGAZZINARE CORRENTE
                              IN SUCCESSIVA
         SET   FLAG, H        FLAG SCAMBIATO - 1
NOSWITCH INC   IX             PROSSIMA ENTRATA
         DJNZ  NEXT           DECREMENTA B, CONTINUA
                              FINO A ZERO
         BIT   FLAG, H        SCAMBIATO = 1?
         JR    NZ, AGAIN      RIPETI SE FLAG = 1
         RET
```

SOMMARIO

In questo capitolo sono state presentate routine di utilità di largo impiego che usano le combinazioni delle tecniche che abbiamo descritto nei capitoli precedenti. Molte di queste routine usano una struttura dei dati speciale, la tabella. Esistono altre possibilità per strutturare i dati ed ora saranno esaminate.

CAPITOLO 9

STRUTTURE DEI DATI

PARTE I — TEORIA

INTRODUZIONE

Il progetto di un buon programma implica due lavori: il *progetto dell'algoritmo* e il *progetto delle strutture dei dati*.

Nella maggior parte dei programmi più semplici, non sono implicate strutture dei dati significative, perciò l'obiettivo principale per imparare la programmazione è di progettare algoritmi e codificarli in modo efficiente in un dato linguaggio di macchina. Questo è ciò che è stato fatto fino ad ora.

Comunque, per progettare programmi più complessi si richiede anche una comprensione delle strutture dei dati. Nel libro sono già state usate due strutture dei dati: la tabella e lo stack.

Lo scopo di questo capitolo è di presentare altre strutture dati, più generali, che potreste voler usare. Questo capitolo è completamente indipendente dal microprocessore, o anche il computer, scelto. È teorico e implica l'organizzazione logica dei dati nel sistema. Esistono libri specializzati sulle strutture dei dati, proprio come esistono libri specializzati sulla moltiplicazione, divisione o di altri algoritmi soliti. Perciò, questo capitolo sarà limitato solo ai punti essenziali. Non ha pretese di essere completo.

Ora saranno esaminate le strutture dei dati più comuni.

PUNTATORI

Un puntatore è un numero che viene impiegato per designare la posizione del dato reale. Ogni puntatore è un indirizzo. Comunque, ogni indirizzo non è necessariamente chiamato puntatore. Un indirizzo è un puntatore solo se punta a qualche tipo di dati o ad informazioni strutturate. Abbiamo già incontrato un puntatore tipico: il puntatore dello stack, il quale punta alla parte alta dello stack (o di solito proprio alla cima dello stack). Vedremo che lo stack è una struttura dei dati comune, chiamata struttura LIFO (last-in-first-out cioé il primo elemento che entra è l'ultimo che esce).

Come altro esempio, quando si usa l'indirizzamento indiretto, l'indirizzo indiretto è sempre un puntatore ai dati che si desidera richiamare.

Esercizio 9-1: *Esaminare la Figura 9-1. All'indirizzo 15 nella memoria, c'è un puntatore alla Tabella T. La Tabella T inizia all'indirizzo 500. Qual è il vero contenuto del puntatore a T?*

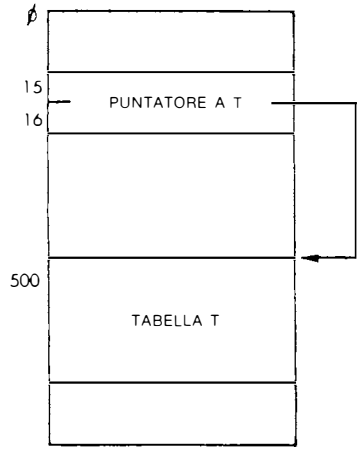


Figura 9-1: Puntatore indirezionale

LISTE

Quasi tutte le strutture dei dati sono organizzate come liste di vario tipo.

Liste sequenziali

Una lista sequenziale, o tabella, o blocco, è probabilmente la struttura dei dati più semplice, ed è quella che abbiamo già usato. Le tabelle sono normalmente ordinate in funzione di un criterio specifico, come la disposizione alfabetica o disposizione numerica. È poi facile richiamare un elemento in una tabella, usando, per esempio, l'indirizzamento indicizzato, come abbiamo fatto. Un blocco normalmente si riferisce ad un gruppo di dati che ha limiti definiti ma il cui contenuto non è ordinato. Può contenere una stringa di caratteri; può essere un settore su un disco; o può essere qualche area logica (chiamata segmento) della memoria. In tali casi, può non essere facile accedere ad un elemento a caso del blocco.

Per facilitare il richiamo di blocchi d'informazioni sono usati i "direttorî".

Direttori

Un direttorio è una lista di tabelle o di blocchi. Per esempio, il sistema a "file" userà normalmente una struttura a direttorio. Come semplice esempio, il direttorio principale del sistema può includere una lista dei nomi di coloro che usano il sistema. Questo è illustrato nella Figura 9-2. L'entrata per l'utente "John" punta al direttorio dei file di John. Il direttorio dei file è una tabella che contiene i nomi di tutti i file di John e le loro posizioni. Questa è, di nuovo, una tabella di puntatori. In questo caso abbiamo costruito un direttorio a due livelli. Un sistema a direttorio flessibile permetterà l'inclusione di direttori intermedi addizionali, come può essere conveniente per l'utente.

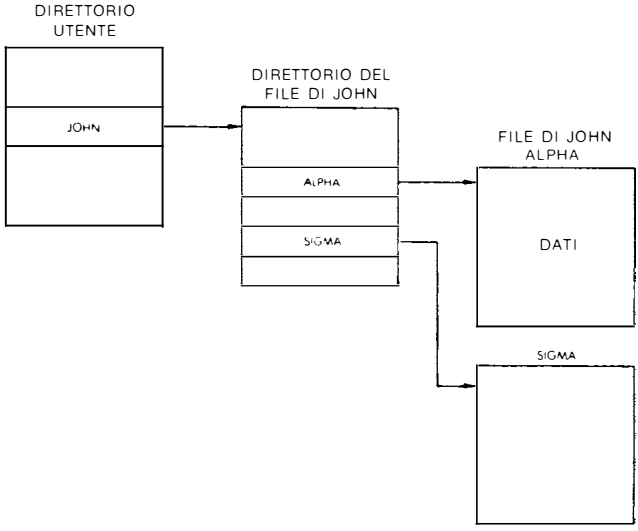


Figura 9-2: Struttura a direttorî

Liste collegate

In un sistema ci sono spesso blocchi di informazioni che rappresentano dati, eventi, o altre strutture che non possono essere facilmente trasferite.

Se fosse possibile, noi probabilmente le assembleremmo in una tabella per classificarle o strutturarle. Il problema adesso è che desideriamo lasciarle dove sono e tuttavia stabilire un ordine tra di esse come prima, seconda, terza, quarta, Per risolvere questo problema sarà usata una lista collegata (linked list). Il concetto di lista collegata è mostrato nella Figura 9-3. Nell'illustrazione, vediamo che un puntatore della lista, chiamato FIRSTBLOCK, punta all'inizio del primo blocco. Una posizione dedicata all'interno del Block 1, come la prima o l'ultima parola in esso, contiene un puntatore al Block 2, chiamato PTR1. Il procedimento è poi ripetuto per il Block 2 e il Block 3. Poiché il Block 3 è l'ultima entrata nella lista, PTR3, per convenzione, o contiene un valore "nullo" speciale, oppure punta a se stesso, in modo che può essere rivelata la fine della lista. Questa struttura è economica, perché richiede solo pochi puntatori (uno per blocco) e libera l'utente dal muovere fisicamente i blocchi nella memoria.

Figura 9-3: Una lista collegata

Per esempio, esaminiamo come sarà inserito un nuovo blocco. Questo è illustrato nella Figura 9-4. Supponiamo che il nuovo blocco sia all'indirizzo NEW BLOCK, e debba essere inserito tra il Block 1 e il Block 2. Il puntatore PTR1 è semplicemente variato al valore NEW BLOCK, in modo che ora punti al Block X. PTRX conterrà il precedente valore di PTR1, cioè, punterà al Block 2. Gli altri puntatori nella struttura sono lasciati invariati. Possiamo vedere che l'inserimento di un nuovo blocco ha richiesto semplicemente l'aggiornamento di due puntatori nella struttura. Questo è chiaramente efficiente.

Esercizio 9-2: *Disegnate un diagramma che mostri come sarebbe rimosso il Block 2 da questa struttura.*

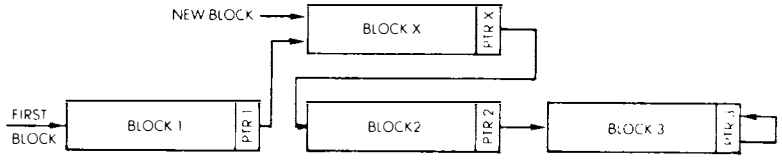


Figura 9-4: Inserimento di un nuovo blocco

Sono stati sviluppati diversi tipi di liste per facilitare tipi specifici di accesso, di inserzioni, e di soppressione nella lista e dalla lista. Esaminiamo alcuni dei tipi più frequentemente usati di liste collegate.

Coda

Una coda è formalmente chiamata FIFO (first-in-first-out cioè il primo elemento che entra è il primo che esce). Una coda è mostrata nella Figura 9-5. Per chiarire lo schema, supponiamo, per esempio, che il blocco a sinistra sia una routine di servizio per un dispositivo d'output, come un stampante. I blocchi che compaiono a destra sono blocchi di richiesta da vari programmi o routine, per stampare caratteri. L'ordine in cui saranno serviti è l'ordine stabilito dalla coda di attesa. Si può osservare che il primo evento che otterrà assistenza è il Block 1, il seguente è il Block 2, e quello dopo ancora è il Block 3. In una coda, la convenzione è che ogni evento che arriva nella coda sarà inserito alla fine. Qui sarà inserito dopo PTR3. Ciò garantisce che il primo blocco ad essere inserito nella coda sarà il primo ad essere servito. In un computer è piuttosto comune avere code di un certo numero di eventi quando essi devono accedere ad una risorsa che risulta scarsa, come il processore o qualche dispositivo d'input/output.

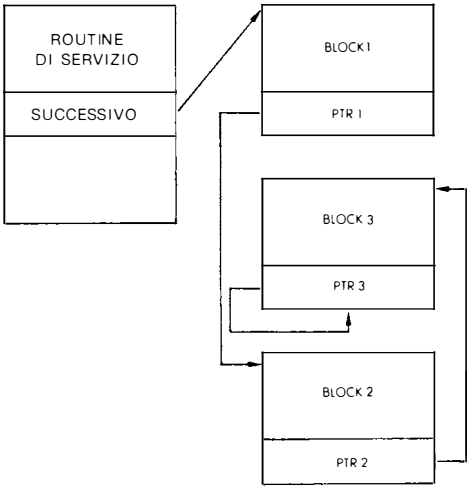


Figura 9-5: Una coda

Stack

La struttura stack è già stata studiata in dettaglio nel corso del libro. È una struttura in cui l'ultimo elemento che entra è il primo che esce (LIFO: last-in-first-out). L'ultimo elemento depositato in alto è il primo ad essere rimosso. Uno stack può essere realizzato come blocco classificato, oppure può essere realizzato come lista. Siccome la maggior parte degli stack nei microprocessori sono usati per eventi ad alta velocità, come le subroutine e gli interrupt, di solito è assegnato allo stack un blocco continuo invece di usare una lista collegata.

Confronto fra lista collegata e blocco

Allo stesso modo, la coda potrebbe essere realizzata come un blocco di locazioni riservate. Il vantaggio di usare un blocco continuo è il reperimento veloce e l'eliminazione dei puntatori. Lo svantaggio è quello che di solito è necessario dedicare un blocco piuttosto grande per alloggiare la massima dimensione prevista della struttura. Inoltre, questo metodo rende anche difficile e poco pratico inserire o rimuovere elementi dall'interno del blocco. Dal momento che la memoria è tradizionalmente una risorsa scarsa, i blocchi sono stati di solito riservati per strutture a misura fissa o per strutture richiedenti la velocità massima di reperimento, come lo stack.

Lista circolare

"Round robin" è un nome comune per una lista circolare. Una lista circolare è una lista collegata in cui l'ultima entrata punta indietro alla prima. Questo è illustrato nella Figura 9-6. Nel caso di una lista circolare, è spesso tenuto un puntatore al *blocco corrente*. Nel caso di eventi, o programmi, che aspettano servizio, il puntatore all'*evento corrente*, sarà, ogni volta spostato di una posizione a sinistra o a destra. Di solito un "round robin" corrisponde ad una struttura in cui tutti i blocchi si presume abbiano la stessa priorità. Comunque, una lista circolare può anche essere usata come un sotto caso di altre strutture semplicemente per facilitare il reperimento del primo blocco dopo l'ultimo, quando si esegue una ricerca.

Come esempio di una lista circolare, un programma di polling di solito va in una maniera "round robin", interrogando tutti i periferici e poi ritornando al primo.

Alberi

Può essere usata una struttura ad albero ogni qual volta esiste una relazione logica tra tutti gli elementi di una struttura (questa di solito è chiamata una sintassi). Un esempio semplice di una struttura ad albero è un albero discendente, o genealogico.

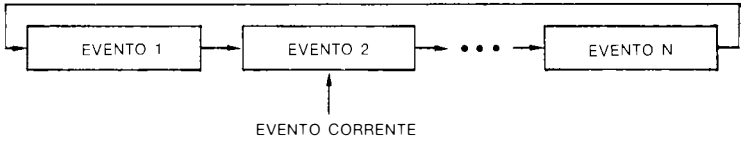


Figura 9-6: Il "Round Robin" è una lista circolare

Questo è illustrato nella Figura 9-7. Si può vedere che Smith ha due figli: un figlio, Robert, e una figlia, Jane. Jane a sua volta ha tre figli, Liz, Tom e Phil. Tom a sua volta ha altri due bambini: Marx e Chris. Inoltre, Robert, alla sinistra dell'illustrazione, non ha discendenti.

Questo è un albero strutturato. Infatti, abbiamo già incontrato un esempio di albero semplice nella Figura 9-2. La struttura a direttorio è un albero a due livelli. Gli alberi sono usati vantaggiosamente ogni qual volta gli elementi possono essere classificati secondo una struttura fissa. Inoltre, possono stabilire gruppi d'informazioni in una maniera strutturata che può essere richiesta per le elaborazioni successive, come in un progetto di un compilatore o di un interprete.

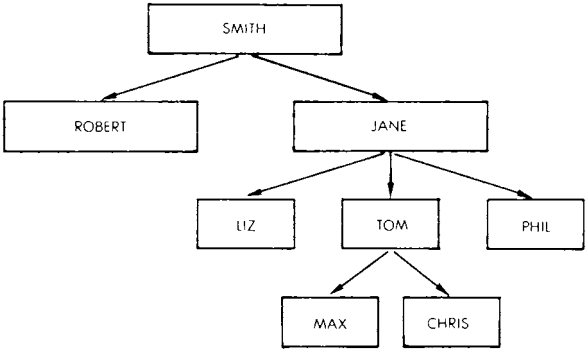


Figura 9-7: Albero genealogico

Liste doppiamente collegate

Possono essere stabiliti collegamenti addizionali tra gli elementi di una lista.

L'esempio più semplice è la lista doppiamente collegata. Questo è illustrato nella Figura 9-8. Possiamo vedere che abbiamo la solita sequenza di collegamenti da sinistra a destra, più un'altra sequenza di collegamenti da destra a sinistra. Lo scopo è quello di permettere un facile reperimento dell'elemento precedente quello che sta per essere elaborato, così come quello seguente. Questo costa un puntatore extra per blocco.

Figura 9-8: Lista doppiamente collegata

RICERCA E CLASSIFICAZIONE

La ricerca e la classificazione di elementi di una lista dipende direttamente dal tipo di struttura che è stata usata per la lista. Molti algoritmi di ricerca sono stati sviluppati per le strutture dati più frequentemente usate. Abbiamo già usato l'indirizzamento indicizzato.

Questo è possibile ogni qual volta gli elementi di una tabella sono ordinati in funzione di un criterio noto. Tali elementi possono essere reperiti tramite i loro numeri.

La *ricerca sequenziale* si riferisce alla scansione lineare di un intero blocco. Questo è chiaramente inefficiente ma può essere usato quando non c'è nessuna altra tecnica migliore, o per mancanza di ordine degli elementi.

La *ricerca binaria*, o *logaritmica* cerca un elemento in una lista classificata dividendo l'intervallo di ricerca a metà ad ogni fase. Supponendo che noi stiamo cercando una lista alfabetica, si potrebbe iniziare, per esempio, a metà di una tabella e determinare se il nome che stiamo cercando è prima o dopo questo punto. Se è dopo questo punto, elimineremo la prima metà della tabella e guarderemo all'elemento intermedio della seconda metà. Paragoniamo questa entrata nuovamente con quella che stiamo cercando, e restringiamo la nostra ricerca ad una delle due metà, e così via. La lunghezza massima di una ricerca è poi garantita essere $\log_2 n$. dove n è il numero di elementi nella tabella.

Esistono molte altre tecniche di ricerca.

SOMMARIO DEL CAPITOLO

Questo capitolo era inteso solo come una breve presentazione di strutture dei dati solite che possono essere usare da un programmatore.

Sebbene le strutture dei dati più comuni sono state organizzate in tipi ed è stato dato loro un nome, l'organizzazione globale dei dati in un sistema complesso può usare qualsiasi combinazione di esse, oppure richiedere al programmatore d'inventare strutture più appropriate. La schiera delle possibilità è limitata soltanto dalla immaginazione del programmatore. Allo stesso modo, è stato sviluppato un numero di tecniche ben note di classificazione e di ricerca per far fronte alle solite strutture dei dati. Una descrizione completa è oltre la portata del libro. Il contenuto di questo capitolo intendeva porre l'accento sull'importanza di progettare appropriate strutture per i dati che devono essere manipolati e di fornire gli strumenti fondamentali per raggiungere lo scopo. Ora saranno presentati in dettaglio degli esempi di programmazione.

PARTE II — ESEMPI DI PROGETTO

INTRODUZIONE

Qui saranno presentati veri esempi di progetto delle strutture di dati tipiche: tabella, lista classificata, lista collegata. Per queste strutture saranno programmati algoritmi di ricerca, inserzione e cancellazione. Il lettore interessato a queste tecniche di programmazione avanzate è incoraggiato ad analizzare dettagliatamente i programmi presentati in questo capitolo.

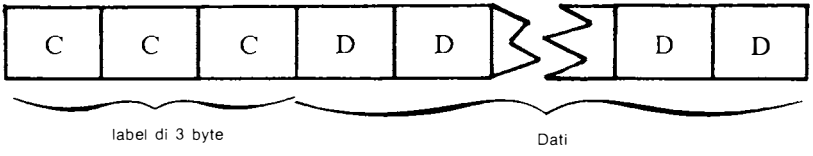
Comunque, il programmatore alle prime armi può inizialmente saltare questo capitolo, e ritornarci quando si sente pronto.

Per seguire gli esempi di progetto è necessaria una buona comprensione dei concetti presentati nella prima parte di questo capitolo. I programmi useranno anche tutti i modi d'indirizzamento dello Z80, e integreranno molti dei concetti e delle tecniche presentate nei capitoli precedenti.

Ora saranno introdotte tre strutture: una lista semplice, una lista alfabetica ed una lista collegata più il direttorio. Per ogni struttura, saranno sviluppati tre programmi: ricerca, inserzione e cancellazione.

RAPPRESENTAZIONE DEI DATI PER LA LISTA

Sia la lista semplice che la lista alfabetica useranno una rappresentazione comune per ogni elemento della lista:



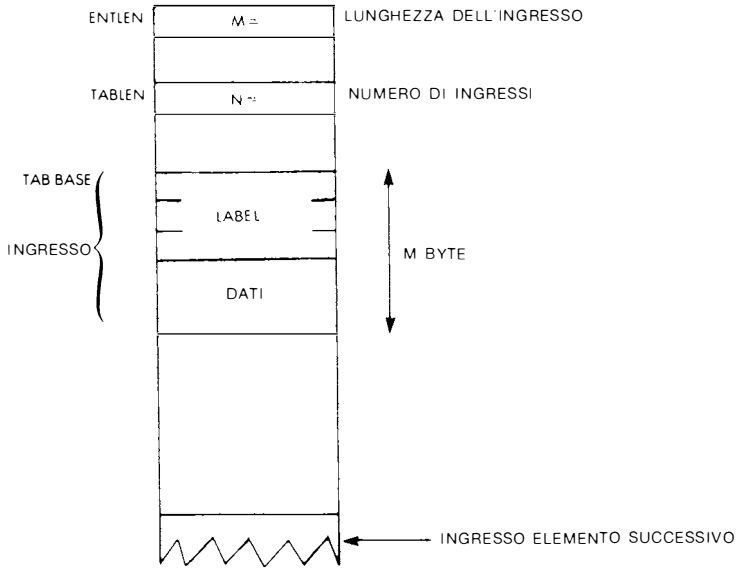


Figura 9-9: Struttura della Tabella

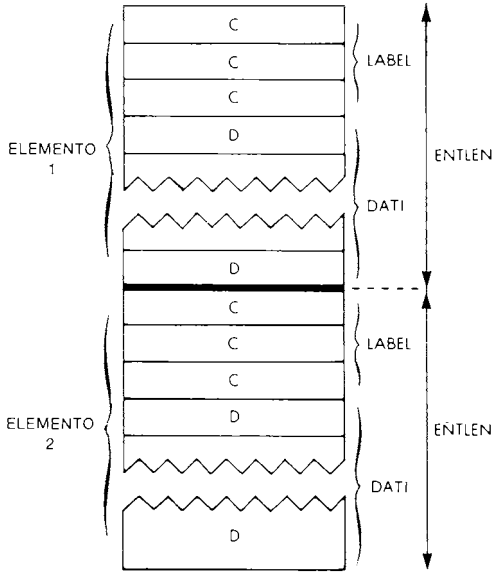


Figura 9-10: Tipica lista di ingressi in memoria

Ogni elemento, o ingresso, include un label (etichetta) di 3 byte, ed un blocco di n byte di dati, con n compreso tra 1 e 253. Perciò, al massimo, ogni ingresso usa una pagina (256 byte). All'interno di ogni lista, tutti gli elementi hanno la stessa lunghezza (vedere la Figura 9-10). I programmi operanti su queste due liste semplici usano alcune convenzioni sulle variabili comuni:

ENTLEN è la lunghezza di un elemento. Per esempio, se ogni elemento ha 10 byte di dati, ENTLEN = 3 + 10 = 13.
TABASE è la base della lista o tabella nella memoria.
POINTR è un puntatore continuo all'elemento corrente.
OBJECT è l'ingresso corrente che deve essere sistemato, inserito o cancellato.
TABLEN è il numero degli ingressi.

Si suppone che tutte le label siano distinte. La variazione di questa convenzione richiederebbe una variazione minore nei programmi.

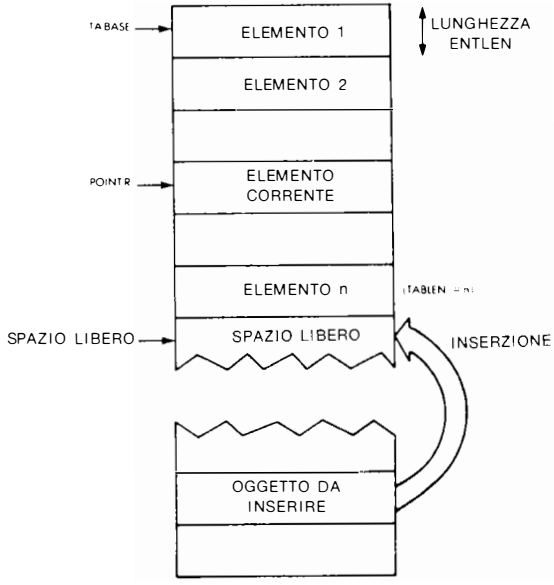


Figura 9-11: La lista semplice

UNA LISTA SEMPLICE

La lista semplice è organizzata come una tabella di n elementi. Gli elementi non sono classificati (vedere la Figura 9-11). Quando si ricerca, si deve esplorare la lista fino a quando non viene trovato un ingresso oppure viene raggiunta la fine della tabella. Quando si inserisce, sono aggiunti nuovi ingressi a quelli già esistenti. Quando è cancellato un ingresso, gli ingressi nelle posizioni di memoria più alte, se ce ne sono, saranno spostati in basso per mantenere continua la tabella.

Ricerca

Si usa una tecnica di ricerca in serie. Ogni campo della label dell'ingresso è confrontato alternativamente alla label di OBJECT, lettera per lettera.

Il puntatore POINTR è inizializzato al valore di TABASE.

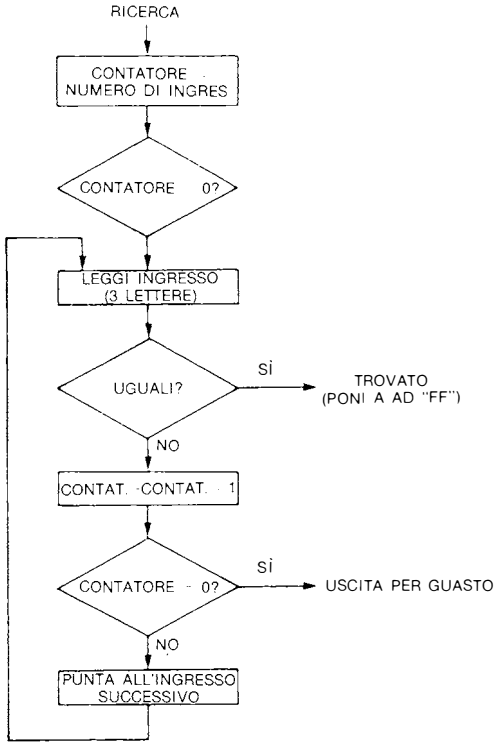


Figura 9-12: Diagramma di flusso della ricerca nella Tabella

Il registro indice X è inizializzato al numero di ingressi contenuto nella lista (immagazzinati a TABLEN). La ricerca procede nel modo ovvio, e il corrispondente diagramma di flusso è mostrato nella Figura 9-12. Il programma compare nella Figura 9-16 alla fine di questo capitolo (programma "SEARCH"). Una prova campione del programma è mostrata nella Figura 9-17.

Inserzione

Quando si inserisce un nuovo elemento, viene impiegato il primo blocco di byte di memoria disponibile (ENTLEN) alla fine della lista (vedere la Figura 9-11).

Il programma per prima cosa controlla che il nuovo ingresso non sia già nella lista (in questo esempio si suppone che tutte le label siano distinte). Se non lo sono, incrementa la lunghezza della lista TABLEN, e muove l'OBJECT alla fine della lista. Nella Figura 9-13 è mostrato il corrispondente diagramma di flusso.

Il programma è mostrato nella Figura 9-16. È chiamato "NEW" e risiede nelle locazioni di memoria da 0135 a 015E.

Il registro indice IY punta alla sorgente. HL e DE sono i puntatori alla destinazione.

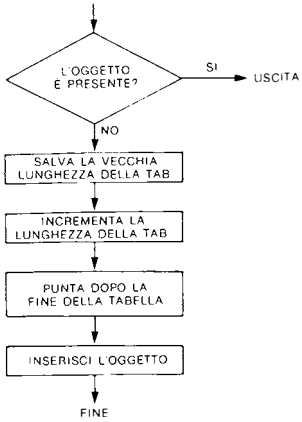


Figura 9-13: Diagramma di flusso dell'inserzione in tabella

Cancellazione

Per cancellare un elemento dalla lista, gli elementi che lo seguono nella lista ad indirizzi più alti sono semplicemente spostati in alto di una posizione di elemento. La lunghezza della lista è decrementata. Questo è illustrato nella Figura 9-14.

Il corrispondente programma è immediato e compare nelle Figure 9-16. È chiamato "DELETE", e risiede agli indirizzi di memoria da 015F a 0187. Il diagramma di flusso è mostrato nella Figura 9-15.

La locazione di memoria TEMPTR è usata come un puntatore temporaneo che punta all'elemento che deve essere spostato in alto.

Durante il trasferimento, POINTR punta sempre al "buco" nella lista, cioè, la destinazione del prossimo blocco che viene trasferito.

Il flag Z è usato per indicare che la cancellazione dell'uscita ha avuto successo.

Notate come l'istruzione LDIR sia usata per l'efficiente trasferimento dei blocchi automatizzato (riferitevi all'indirizzo 0178 nella Figura 9-16).

```
        LD      A, B            CONTATORE DEI BLOCCHI
NEWBLOC LD      BC, (ENTLEN)    LUNGHEZZA DEI BLOCCHI
        LDIR
        DEC     A
        JP      NZ, NEWBLOC
```

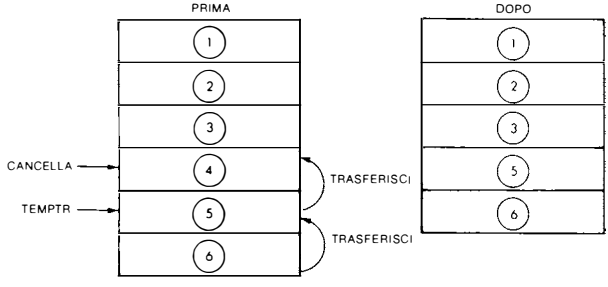


Figura 9-14: Cancellazione di un'entrata (lista semplice)

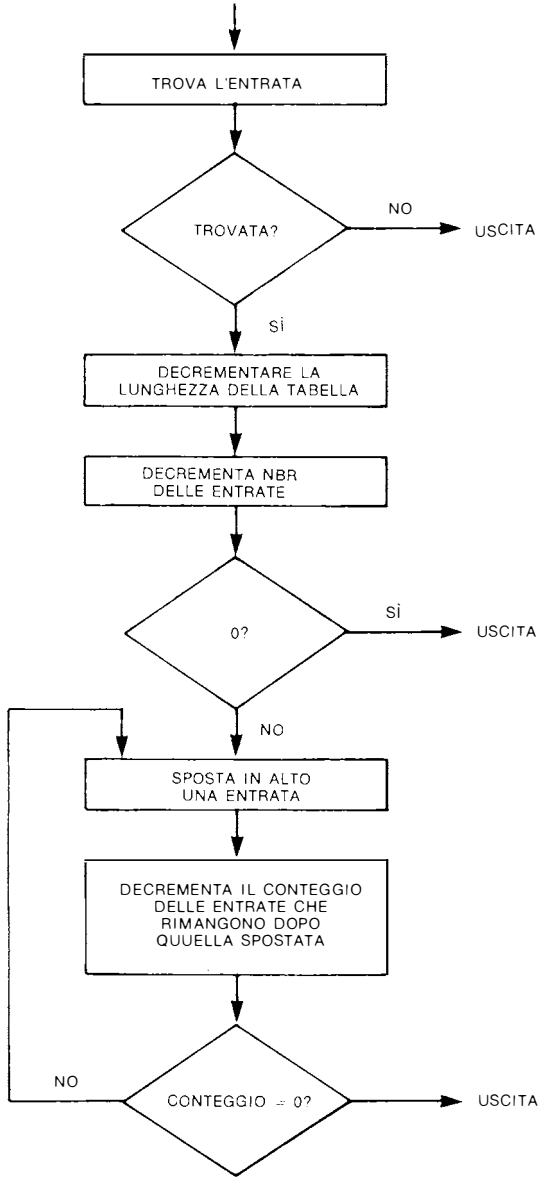


Figura 9-15: Diagramma di flusso della cancellazione in tabella

```
0000                        ORG   0100H
      (0187)   ENTLEN       DL    ENDER
      (0189)   TABLEN       DL    ENDER+2
      (018A)   TABASE       DL    ENDER+3
      (018C)   TEMP         DL    ENDER+5
               ;
0100  1600     SEARCH       LD    D, 0          ;CANCELLA D
0102  3A8901                LD    A, (TABLEN)   ;VERIFICA UNA LUNGHEZZA ZERO DELLA TABELLA
0105  A7                    AND   A             ;POSIZIONA I FLAG
0106  C8                    RET   Z
0107  47                    LD    B, A          ;IMMAGAZZINA LA LUNGHEZZA DELLA TABELLA
0108  DD2A8A01              LD    IX, (TABASE)  ;CARICA L'INDIRIZZO DI BASE IN IX
010C  DD7E00   LOOP         LD    A, (IX+0)     ;VERIFICA LA PRIMA LETTERA DI ENTRATA
010F  FDBE00                CP    (IY+0)
0112  C22701                JP    NZ, NEXTONE
0115  DD7E01                LD    A, (IX+1)     ;VERIFICA LA SECONDA LETTERA
0118  FDBE01                CP    (IY+1)
011B  C22701                JP    NZ, NEXTONE
011E  DD7E02                LD    A, (IX+2)     ;VERIFICA LA TERZA LETTERA
0121  FDBE02                CP    (IY+2)
0124  CA3201                JP    Z, FOUND      ;ESCI SE TUTTE LE LETTERE SI ADATTANO
0127  05       NEXTONE      DEC   B             ;DECREMENTA IL CONTATORE DI LUNGHEZZA DELLA TABELLA
0128  C8                    RET   Z             ;ESCI SE ALLA FINE DELLA TABELLA
0129  ED5B8701              LD    DE, (ENTLEN)  ;PONI IX AL PROSSIMO INDIRIZZO DI ENTRATA.
012D  DD19                  ADD   IX, DE
012F  C30C01                JP    LOOP          ;PROVA DI NUOVO
0132  16FF     FOUND        LD    D, 0FFH       ;PONI D PER MOSTRARE CHE IX CONTIENE L'INDIRIZZO
0134  C9                    RET                 ;DI ENTRATA IN TABELLA
               ;
               ;
               ;
0135  CD0001   NEW          CALL  SEARCH        ;VEDI SE C'È L'OGGETTO
0138  14                    INC   D
0139  CA5E01                JP    Z, OUTE       ;SE D ERA FF, ESCI
013C  3A8901                LD    A, (TABLEN)
013F  5F                    LD    E, A          ;CARICA E CON LA LUNGHEZZA DELLA TABELLA
0140  3C                    INC   A
0141  328901                LD    (TABLEN), A   ;INCREMENTA LA LUNGHEZZA DELLA TABELLA
0144  1600                  LD    D, 0
0146  2A8A01                LD    HL, (TABASE)
0149  ED4B8701              LD    BC, (ENTLEN)  ;PONI B ALLA LUNGHEZZA DI UN'ENTRATA
014D  41                    LD    B, C
014E  19       LOOPE        ADD   HL, DE
014F  10FD                  DJNZ  LOOPE         ;SOMMA HL A (ENTLEN X TABLEN)
0151  ED4B8701              LD    BC, (ENTLEN)
0155  FDE5                  PUSH  IY            ;TRASFERISCI IY A DE
0157  D1                    POP   DE
0158  EB                    EX    DE, HL
0159  EDB0                  LDIR                ;TRASFERISCI MEMORIA DA OGGETTO ALLA FINE
015B  01FFFF                LD    BC, 0FFFFH    ;...DELLA TABELLA
015E  C9       OUTE         RET
               ;
               ;
               ;
015F  CD0001   DELETE       CALL  SEARCH        ;TROVA L'ENTRATA DA CANCELLARE
0162  14                    INC   D             ;VEDI SE TROVATA
0163  C28601                JP    NZ, OUT
0166  3A8901                LD    A, (TABLEN)   ;DECREMENTA LA LUNGHEZZA DELLA TABELLA
0169  3D                    DEC   A
016A  328901                LD    (TABLEN), A
016D  05                    DEC   B             ;B ORA = NUMERO DI ENTRATE LASCIATE IN TABELLA
016E  CA8301                JP    Z, EXIT       ;... DOPO QUELLA DA CANCELLARE
0171  DDE5                  PUSH  IX            ;TRASFERISCI IX A DE
```

Figura 9-16: Lista semplice — I programmi

```
0173   D1                    POP    DE
0174   2A8701                LD     HL, (ENTLEN)     ;PONI HL DI UN'ENTRATA AVANTI RISPETTO A QUELLA DI DE
0177   19                    ADD    HL, DE
0178   78                    LD     A, B             ;POSIZIONA IL CONTATORE DI BLOCCO
0179   ED4B8701   NEWBLOC   LD     BC, (ENTLEN)     ;POSIZIONA IL CONTATORE DI LUNGHEZZA DI BLOCCO
017D   EDB0                 LDIR                    ;TRASFERISCI L'ENTRATA 1 DELLA TABELLA
017F   3D                   DEC    A
0180   C27901               JP     NZ, NEWBLOC      ;TRASFERISCI UN ALTRO BLOCCO
0183   01FFFF     EXIT      LD     BC, OFFFFH       ;MOSTRA CHE È STATO ESEGUITO
0186   C9         OUT       RET
                  ;
0187   (0000)     ENDER     END

SYMBOL TABLE

DELETE  015F   ENDER   0187   ENTLEN  0187   EXIT    0183   FOUND   0132
LOOP    010C   LOOPE   014E   NEW     0135   NEWBLO  0179   NEXTON  0127
OUT     0186   OUTE    015E   SEARCH  0100   TABASE  018A   TABLEN  0189
TEMP    018C
```

Figura 9-16: Lista semplice — I programmi (continua)

Visualizzazione Memoria

Elenco degli oggetti con la loro locazione in memoria

```
-DM300
0300  53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 00 00  SON111111111111...
0310  44 41 44 32 32 32 32 32 32 32 32 32 32 00 00 00  DAD2222222222222...
0320  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 00 00  MOM3333333333333...
0330  55 4E 43 34 34 34 34 34 34 34 34 34 34 00 00 00  UNC4444444444444...
0340  41 4E 54 35 35 35 35 35 35 35 35 35 35 00 00 00  ANT5555555555555...
0350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0360  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0370  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

-SY
Y=0000 300
```

Poni IY a 0300H (puntatore a OBJECT)

```
-G193/196
P=0196 0196
```

Esecuzione di 'INSERT'

Configurazione della tabella dopo lo svolgimento del programma

```
-DM400
0400  53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 00 00  SON111111111111...
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

-SY
Y=0300 310
```

Poni IY a 0310H (prossimo OBJECT)

```
-G193/196
P=0196 0196
```

Esecuzione di 'INSERT'

Configurazione della tabella dopo la seconda inserzione

```
-DM400
0400  53 4F 4E 31 31 31 31 31 31 31 31 31 31 31 44 41 44  SON1111111111DAD
0410  32 32 32 32 32 32 32 32 32 32 32 00 00 00 00 00 00  2222222222......
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figura 9-17: Lista semplice — Una prova di Esecuzione

```
                    . . .  (Altre inserzioni)  . . .
-DM400
0400  53 4F 4E 31 31 31 31 31 31 31 31 31 31 44 41 44   SONI111111111DAD
0410  32 32 32 32 32 32 32 32 32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 4D 4F 4D 33 33 33 33 33 33   4444444MOM333333
0430  33 33 33 33 41 4E 54 35 35 35 35 35 35 35 35 35   3333ANT555555555
0440  35 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Configurazione della tabella dopo diverse inserzioni

```
-SY
Y=0340 320
 G190/193

P=0193 0193'
```

Esecuzione di 'SEARCH'

Il registro D mostra che è stato trovato l'oggetto

Contenuto del registro

```
-DR
Z  N   A=4D BC 0PFI DE=FF0D HL 034D S 0100 P 0193 0193' CALL 0135
       A' 00 B'=0000 D' 0000 H'=0000 X 0422 Y 0320 I=00        (0135')
```

Indirizzo di Object

```
-G196/199

P=0199 0199'
```

Esecuzione di 'DELETE'

Configurazione della tabella dopo la cancellazione

```
-DM400
0400  53 4F 4E 31 31 31 31 31 31 31 31 31 31 44 41 44   SONI111111111DAD
0410  32 32 32 32 32 32 32 32 32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 41 4E 54 35 35 35 35 35 35   4444444ANT555555
0430  35 35 35 35 41 4E 54 35 35 35 35 35 35 35 35 35   5555ANT555555555
0440  35 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
-SY
Y=0240 340
 G196/199

P=0199 0199'
```

Cancella l'ultima entrata della tabella

Nota: nessuna apparente variazione nella configurazione della tabella

```
-DM400
0400  53 4F 4E 31 31 31 31 31 31 31 31 31 31 44 41 44   SONI111111111DAD
0410  32 32 32 32 32 32 32 32 32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 41 4E 54 35 35 35 35 35 35   4444444ANT555555
0430  35 35 35 35 41 4E 54 35 35 35 35 35 35 35 35 35   5555ANT555555555
0440  35 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
-DM19951
0189  03
-G190/193
```

Locazione di memoria 'TABLEN' — mostra la vera lunghezza della tabella

```
P=0193 0193'
```

Esecuzione di 'SEARCH' per l'oggetto cancellato

D mostra che non è stato trovato l'oggetto

```
-DR
Z  N   A 55 BC=00FF DE=000D HL=0441 S=0100 P 0193 0193' CALL 0135
       A'=00 B'=0000 D'=0000 H'=0000 X=041A Y 0340 I=00        (0135')
```

Figura 9-17: Lista semplice — Una prova di Esecuzione (continua)

LISTA ALFABETICA

La lista alfabetica, o "tabella", a differenza della precedente, conserva tutti i suoi elementi classificati in ordine alfabetico. Questo permette l'uso di tecniche di ricerca più veloci di quello lineare.

Qui è usata una ricerca binaria.

Ricerca

L'algoritmo di ricerca è una ricerca binaria classica. Ricordiamo che la tecnica è essenzialmente analoga a quella usata per trovare un nome nell'elenco telefonico.

Si inizia da qualche parte nel mezzo dell'elenco, e poi, a secondo degli ingressi trovati, si va avanti o indietro per trovare l'ingresso desiderato. Questo metodo è veloce e ragionevolmente semplice da compiere.

Il diagramma di flusso della ricerca binaria è mostrato nella Figura 9-18 ed il programma è mostrato nella Figura 9-23.

Il secondo flag usato dal programma è CLOSE. Questo flag è posto uguale a COMPRESS quando l'incremento della ricerca INCMNT diventa uguale ad "1". Rivelerà il fatto che l'elemento non è stato trovato se COMPRES non è uguale a CLOSE.

Altre variabili usate dal programma sono:

LOGPOS che indica la posizione logica nella tabella (numero dell'elemento).
INCMNT che rappresenta il valore di cui sarà decrementato o incrementato il puntatore corrente se non riesce il prossimo confronto.
TABLEN come al solito rappresenta la lunghezza totale della lista.

LOGPOS e INCMNT saranno confrontati a TABLEN per assicurare che non siano superati i limiti delle liste.

Il programma chiamato "SEARCH" è mostrato nella Figura 9-23. Risiede nelle posizioni di memoria da 0100 a 01CF e merita d'essere studiato con attenzione, siccome è molto più complesso che nel caso di una ricerca lineare.

Una complicazione addizionale è dovuta al fatto che l'intervallo di ricerca a volte può essere o pari o dispari. Quando è dispari, deve essere introdotta una correzione. (Per esempio, non può puntare all'elemento intermedio di una lista da quattro elementi).

Quando è dispari, viene usato un "trucco" per puntare all'elemento intermedio: la divisione per due è accompagnata da uno spostamento a destra. Il bit "che cade" nel riporto dopo l'istruzione SRL sarà "1" se l'intervallo era dispari. Questo poi è semplicemente sommato al puntatore.

L'OBJECT è poi confrontato all'ingresso centrale del nuovo intervallo di ricerca. Se il confronto è positivo il programma esce. In caso contrario ("NOGOOD"), il riporto è posto a "0" se l'OBJECT è inferiore all'ingresso. Ogni qual volta l'INCMNT diventa "1", il flag CLOSE (che era stato inizializzato a "0" è allora controllato per vedere se era posto ad 1. Se non lo era, lo diventa. Se era posto ad 1 è compiuto un controllo per determinare se abbiamo passato la posizione dove l'OBJECT avrebbe dovuto essere ma non è

```
(0121)    LD    A, C
          SRL   A
          ADC   O
          LD    C, A
```

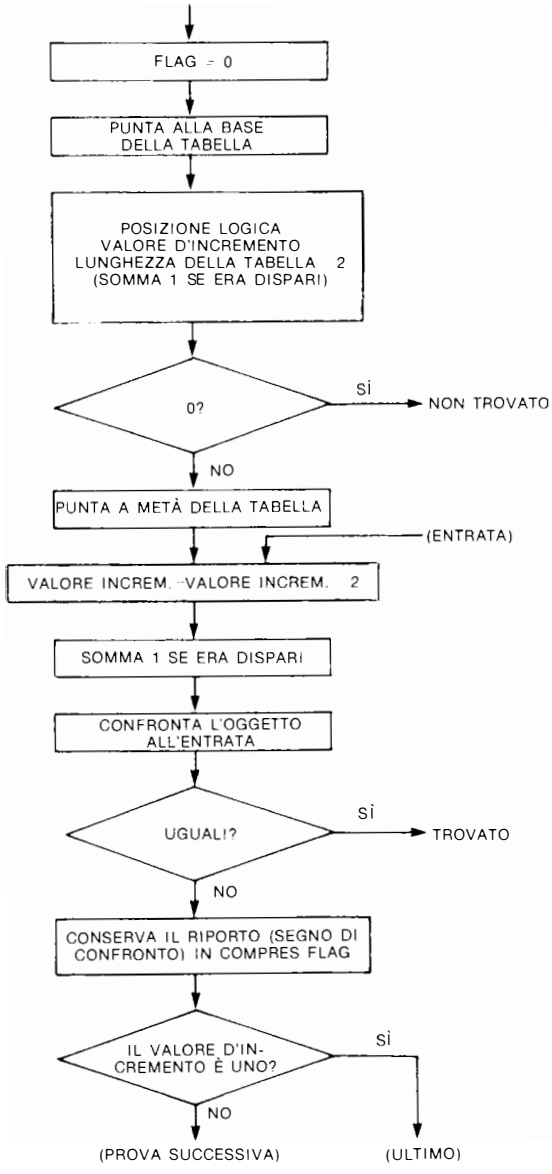


Figura 9-18: Diagramma di flusso della ricerca binaria

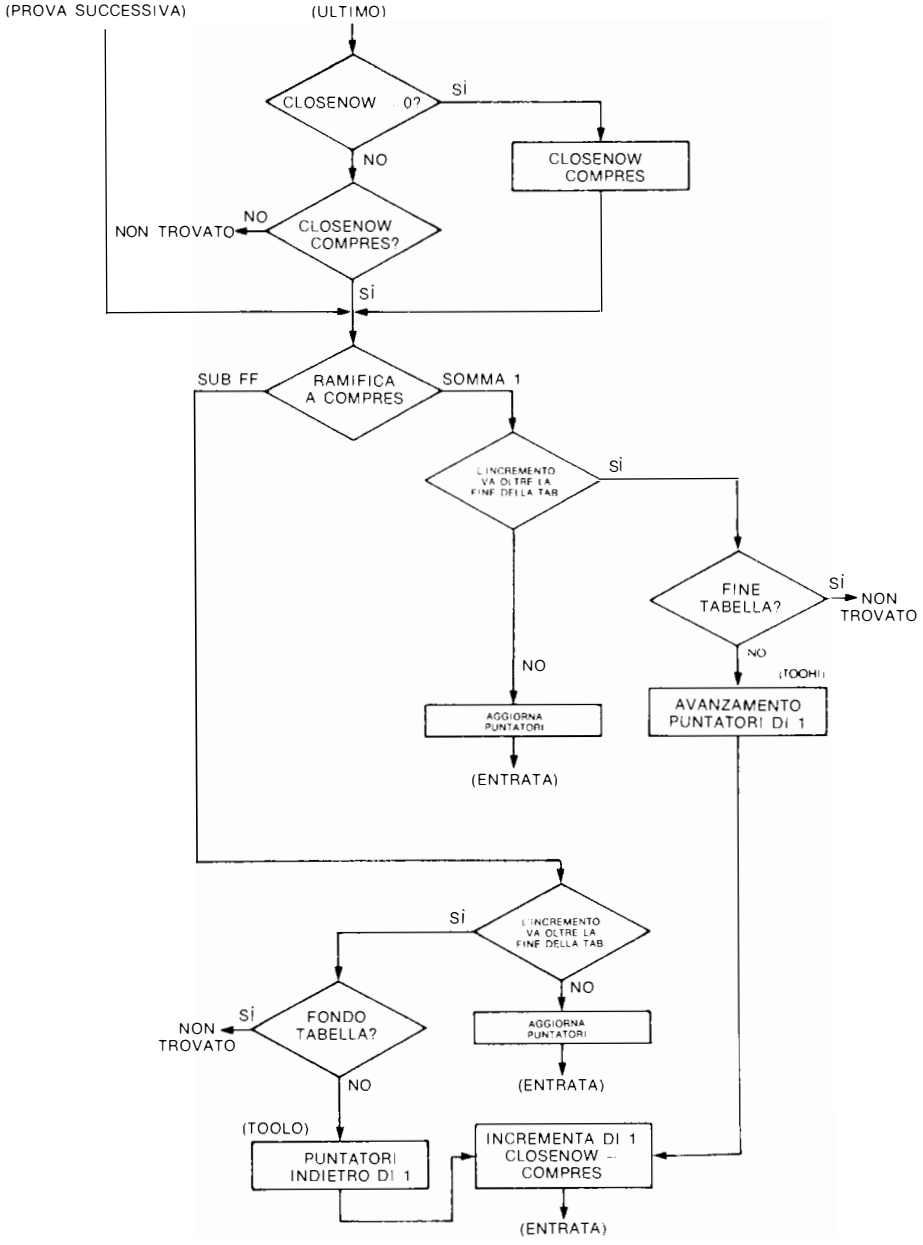


Figura 9-18: Diagrammi di flusso della ricerca binaria (continua)

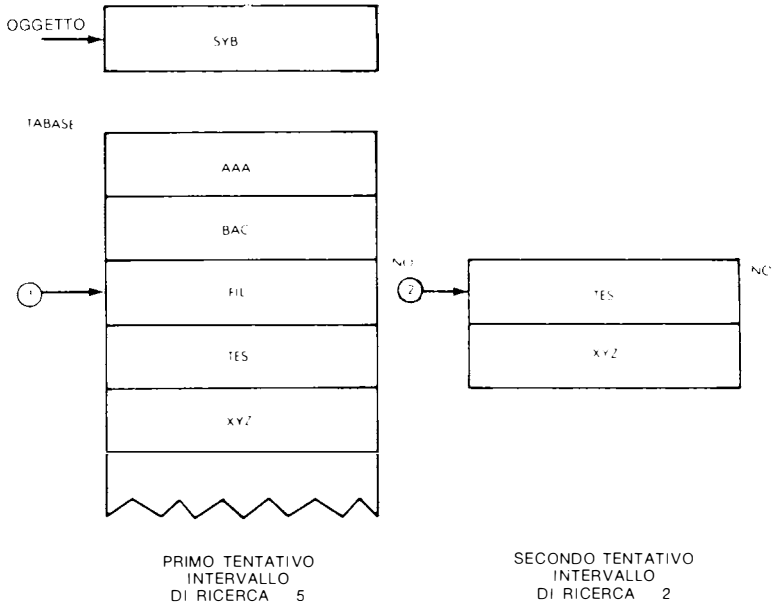


Figura 9-19: Una ricerca binaria

Questa lista conserva gli ingressi in ordine alfabetico e li rintraccia usando una ricerca binaria o "logaritmica", Nella Figura 9-19 è mostrato un esempio. La ricerca in qualche modo è complicata dalle necessità di tener conto di diverse condizioni. Il problema maggiore che deve essere evitato è quello di ricercare un oggetto che non è là. In tale caso, gli ingressi con valori alfabetici immediatamente più alti e più bassi potrebbero essere alternativamente provati per sempre. Per evitare questo, è mantenuto un flag nel programma per conservare il valore del flag di riporto dopo un paragone non riuscito. Quando il valore INCMONT, il quale mostra di quanto il puntatore sarà poi incrementato raggiunge un valore di "1", un altro flag chiamato "CLOSENOW", che noi abbrevieremo a "CLOSE", è posto al valore del flag COMPRES. Perciò, poichè tutti gli ulteriori incrementi saranno "1", se il puntatore va oltre il punto dove dovrebbe essere l'oggetto, COMPRES non eguaglierà più CLOSE e la ricerca terminerà.

Questa caratteristica permette anche alla routine NEW di determinare dove sono localizzati i puntatori logici e fisici, relativi a dove andrà l'oggetto.

Perciò, se l'OBJECT ricercato non è nella tabella, e il puntatore è incrementato di uno, sarà posto ad 1 il flag CLOSE. Nel prossimo passo della routine, il risultato del confronto sarà opposto al precedente. I due flag non si uguaglieranno, e il programma uscirà indicando "not found" (non trovato).

L'altro grosso problema che deve essere trattato è la possibilità di uscire da una estremità della tabella quando si somma o si sottrae il valore d'incremento. Ciò è risolto eseguendo un prova di "somma" o "sottrazione" usando il puntatore logico e il valore di lunghezza che registrano il vero numero di ingressi, non le posizioni fisiche nella memoria usate dai puntatori fisici.

Ricapitolando, sono usati due flag dal programma per memorizzare le informazioni: CMPRES e CLOSE. Il flag CMPRES viene usato per conservare il fatto che il riporto era "0" oppure "1" dopo il confronto più recente. Questo determina se l'elemento sotto test era più grande o più piccolo di quello col quale era confrontato. Il C indica la relazione. Ogni volta che il ri-

porto C era "1", e l'elemento era più piccolo dell'oggetto, CMPRES è posto ad "1". Ogni volta che il riporto C era "0", indicando che l'elemento era più grande dell'oggetto, CMPRES sarà posto ad "FF".

Notate anche che quando il riporto era "1" (un guasto), il puntatore corrente punterà all'ingresso sotto l'OBJECT. Se non lo era, viene posto ad 1. Se era posto ad 1, viene eseguito un controllo per determinare se abbiamo passato la posizione dove l'OBJECT avrebbe dovuto essere ma non è stato trovato.

Inserzione di Elemento

Per inserire un nuovo elemento, viene condotta una ricerca binaria. Se l'elemento è trovato nella tabella non ha bisogno di essere inserito. (Qui supponiamo che tutti gli elementi siano distinti). Se l'elemento non viene trovato nella tabella, deve essere inserito immediatamente prima o immediatamente dopo l'ultimo elemento al quale è stato confrontato. Il valore del flag COMPRES dopo la ricerca indica se dovrebbe essere inserito immediatamente prima o immediatamente dopo. Tutti gli elementi che seguono la nuova posizione dove sta per essere posto sono spostati di una posizione di blocco, ed è inserito il nuovo elemento.

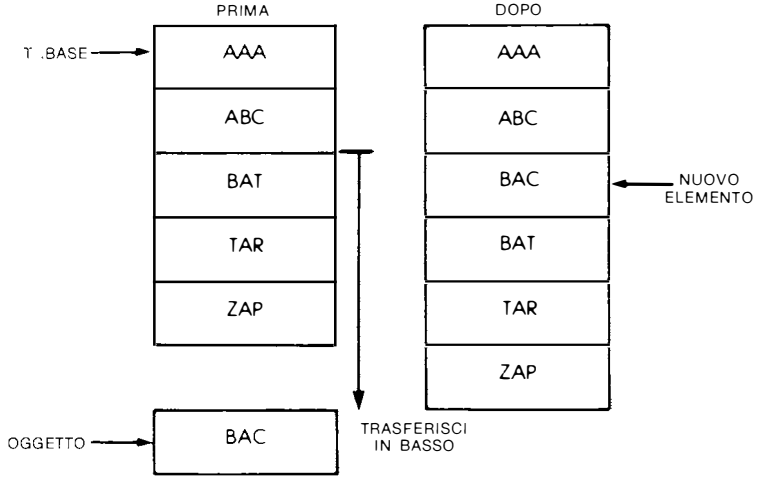


Figura 9-20: Inserzione di "BAC"

Il processo d'inserzione è illustrato nella Figura 9-20, e il programma corrispondente compare nella Figura 9-23.

Il programma è chiamato NEW, ed inizia alla posizione di memoria 01D0. Notate che sono usate le istruzioni dello Z80 automatizzate LDDR e LDIR per trasferimenti efficienti dei blocchi.

Cancellazione di Elemento

Allo stesso modo, per trovare l'oggetto è condotta una ricerca binaria. Se la ricerca non riesce, non ha bisogno di essere trovato. Se la ricerca riesce, l'elemento è cancellato e tutti gli elementi che seguono sono spostati in alto di una posizione di blocco. Un esempio corrispondente è mostrato nella Figura 9-21, e il programma compare nella Figura 9-23. Il diagramma di flusso è mostrato nella Figura 9-22.

Il programma è chiamato "DELETE" e risiede all'indirizzo 0221. Un esempio di "run" dei programmi precedenti è mostrato nella Figura 9-24.

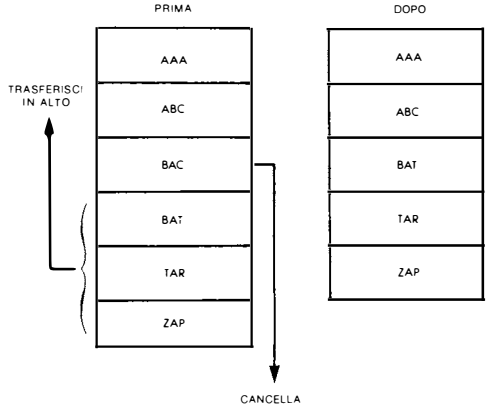


Figura 9-21: Cancellazione di "BAC"

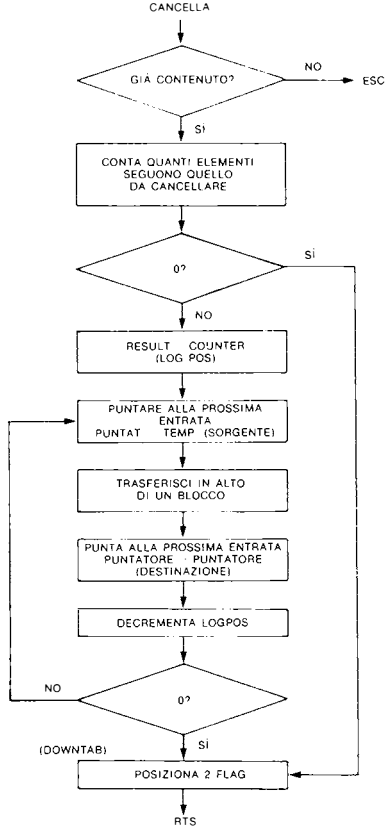


Figura 9-22: Diagramma di flusso di cancellazione (lista alfabetica)

```
0000                        ORG   0100H
     (024A)   CLOSENOW      DL    ENDED
     (024B)   COMPRES       DL    ENDED + 1
     (024C)   TABLEN        DL    ENDED + 2
     (024D)   TABASE        DL    ENDED + 3
     (024F)   ENTLEN        DL    ENDED + 5
              ;
0100 3E00     SEARCH       LD    A, 0
0102 324A02                LD    (CLOSENOW), A   ;LOCAZIONI DEL FLAG ZERO
0105 324B02                LD    (COMPRES), A
0108 57                    LD    D, A
0109 2A4D02                LD    HL, (TABASE)    ;INIZIALIZZA HL
010C 3A4C02                LD    A, (TABLEN)
010F CB3F                  SRL   A               ;DIVIDI PER 2
0111 CE00                  ADC   0               ;SOMMA IL BIT DI 1
0113 4F                    LD    C, A            ;MEMORIZZA COME VALORE DELL'INCREMENTO
0114 47                    LD    B, A            ;MEMORIZZA COME VALORE LOGICO DELLA POSIZIONE
0115 CABA01                JP    Z, NOTFOUND     ;CONTROLLA SE LA LUNGHEZZA È ZERO
0118 5F                    LD    E, A            ;MOLTIPLICA (E- 1) ENTLEN
0119 1D                    DEC   E
011A CDBD01                CALL  MULT
011D 19                    ADD   HL, DE          ;PONI HL PER PUNTARE A METÀ TABELLA
011E E5       ENTRY        PUSH  HL              ;CARICA HL IN IX
011F DDE1                  POP   IX
0121 79                    LD    A, C            ;DIVIDI IL VALORE DELL'INCREMENTO PER DUE
0122 CB3F                  SRL   A
0124 CE00                  ADC   0
0126 4F                    LD    C, A
0127 DD7E00                LD    A, (IX + 0)     ;CONFRONTA LA PRIMA LETTERA
012A FDBE00                CP    (IY + 0)
012D C24201                JP    NZ, NOGOOD
0130 DD7E01                LD    A, (IX + 1)     ;CONFRONTA LA SECONDA LETTERA
0133 FDBE01                CP    (IY + 1)
0136 C24201                JP    NZ, NOGOOD
0139 DD7E02                LD    A, (IX + 2)     ;CONFRONTA LA TERZA LETTERA
013C FDBE02                CP    (IY + 2)
013F CABC01                JP    Z, FOUND
0142 3E01     NOGOOD       LD    A, 1            ;PONI IL FLAG DEL RISULTATO DEL CONFRONTO A
0144 DA4901                JP    C, TESTS        ;...RISULTATO DEL CONFRONTO (1, FF)
0147 3EFF                  LD    A, 0FFH
0149 324B02   TESTS        LD    (COMPRES), A
014C 79                    LD    A, C            ;IL VALORE DELL'INCREMENTO È 1?
014D 3D                    DEC   A
014E C26901                JP    NZ, NEXTEST
0151 3A4A02                LD    A, (CLOSENOW)   ;SÌ, IL FLAG CLOSE È 1?
0154 A7                    AND   A
0155 CA6301                JP    Z, NOTCLOSE
0158 57                    LD    D, A            ;SÌ, VEDI SE SI È ATTRAVERSATO DOVE
0159 3A4B02                LD    A, (COMPRES)    ;...L'ENTRATA DOVREBBE ESSERE MA NON È
015C 92                    SUB   D
015D CA6901                JP    Z, NEXTEST
0160 C3BA01                JP    NOTFOUND
0163 3A4B02   NOTCLOSE     LD    A, (COMPRES)    ;PONI IL FLAG CLOSE ALLA DIREZIONE DI
0166 324A02                LD    (CLOSENOW), A   ;...RICERCA PER EVITARE RIPETIZIONI
0169 DDE5     NEXTEST      PUSH  IX              ;PREPARA HL E DE PER SOMMARE 0
016B E1                    POP   HL                SOTTRAI IL VALORE DELL'INCREMENTO
016C 59                    LD    E, C
016D CDBD01                CALL  MULT
0170 3A4B02                LD    A, (COMPRES)    ;PROVA SE SI DEVE SOMMARE O SOTTRARRE
0173 3C                    INC   A
```

Figura 9-23: Programma di ricerca binaria

```
0174  C29601                JP    NZ, ADDIT
0177  78                    LD    A, B          ;PROVA SE LA SOTTRAZIONE PORTA L'ESECUZIONE
0178  91                    SUB   C               OLTRE IL FONDO DELLA TABELLA
0179  CA8501                JP    Z, TOOLOW
017C  DA8501                JP    C, TOOLOW
017F  47                    LD    B, A          ;PONI IL NUOVO VALORE DELLA POSIZIONE LOGICA
0180  ED52                  SBC   HL, DE        ;CAMBIA L'INDIRIZZO
0182  C31E01                JP    ENTRY
0185  78          TOOLOW    LD    A, B          ;VEDI SE LA POSIZIONE È 1
0186  3D                    DEC   A
0187  CABA01                JP    Z, NOTFOUND   ;SE SÌ, ESCI
018A  ED5B4F02              LD    DE, (ENTLEN)  ;PROPRIO SUB 1 È LA POSIZIONE DELL'ENTRATA
018E  37                    SCF
018F  3F                    CCF
0190  ED52                  SBC   HL, DE
0192  05                    DEC   B             ;CAMBIA LA POSIZIONE LOGICA
0193  C3AF01                JP    REALCLOS
0196  3A4C02      ADDIT     LD    A, (TABLEN)   ;PROVA A VEDERE SE LA POSIZIONE CORRENTE
0199  90                    SUB   B             ;...PIÙ L'INCREMENTO VA OLTRE
019A  91                    SUB   C             ;...LA FINE DELLA TABELLA
019B  DAA501                JP    C, TOOHIGH
019E  19                    ADD   HL, DE        ;SE OK, CAMBIA L'INDIRIZZO EFFETTIVO
019F  78                    LD    A, B          ;CAMBIA IL VALORE DELLA POSIZIONE LOGICA
01A0  81                    ADD   C
01A1  47                    LD    B, A
01A2  C31E01                JP    ENTRY
01A5  81          TOOHIGH   ADD   C             ;VEDI SE LA POSIZIONE È ALLA SOMMITÀ DELLA
01A6  CABA01                JP    Z, NOTFOUND   ;...TABELLA (STESSA DI TABLEN – B)
01A9  ED5B4F02              LD    DE, (ENTLEN)  ;SOMMA UNA POSIZIONE DI ENTRATA
01AD  19                    ADD   HL, DE
01AE  04                    INC   B             ;INCREMENTA LA POSIZIONE LOGICA
01AF  0E01        REALCLOS  LD    C, 1          ;PONI L'INCREMENTO AD 1
01B1  3A4B02                LD    A, (COMPRES)  ;PONI IL FLAG CLOSE PER CONFRONTARE
01B4  324A02                LD    (CLOSENOW), A ;...IL RISULTATO
01B7  C31E01                JP    ENTRY
01BA  16FF        NOTFOUND  LD    B, 0FFH
01BC  C9          FOUND     RET
                  ;
01BD  E5          MULT      PUSH  HL            ;MOLTIPLICA E PER (ENTLEN)
01BE  C5                    PUSH  BC            ;...ALL'USCITA VALORE IN DE
01BF  1600                  LD    D, 0
01C1  210000                LD    HL, 0000
01C4  ED4BF02               LD    BC, (ENTLEN)
01C8  41                    LD    B, C
01C9  19          ADDEM     ADD   HL, DE
01CA  10FD                  DJNZ  ADDEM
01CC  C1                    POP   BC
01CD  EB                    EX    DE, HL
01CE  F1                    POP   HL
01CF  C9                    RET
                  ;
                  ;
                  ;
01D0  CD0001      NEW       CALL  SEARCH        ;VEDI SE L'OGGETTO È QUI
01D3  14                    INC   D
01D4  C22002                JP    NZ, OUT
01D7  3A4C02                LD    A, (TABLEN)   ;CONTROLLO PER TABELLA 0
01DA  A7                    AND   A
01DB  CA0C02                JP    Z, INSERT
01DE  3A4B02                LD    A, (COMPRES)
01E1  3C                    INC   A
01E2  CAED01                JP    Z, HISIDE
```

Figura 9-23: Programma di ricerca binaria (continua)

```
01E5   ED5B4F02              LD     DE, (ENTLEN)     ;COMPRES = 1, PONI HL SUBITO PRIMA DI DOVE
01E9   19                    ADD    HL, DE           ;...DOVREBBE ANDARE L'OGGETTO
01EA   C3EE01                JP     SETUP
01ED   05        HISIDE      DEC    B                ;COMPRES = 0, PONI B PER LA SOTTRAZIONE
01EE   3A4C02    SETUP       LD     A, (TABLEN)      ;VEDI QUANTE ENTRATE RIMANGONO
01F1   90                    SUB    B
01F2   CA0C02                JP     Z, INSERT
01F5   5F                    LD     E, A             ;PONI HL ALL'ULTIMA POSIZIONE NELL'ULTIMA
01F6   CDBD01                CALL   MULT             ;...ENTRATA
01F9   19                    ADD    HL, DE
01FA   2B                    DEC    HL
01FB   EB                    EX     DE, HL           ;PONI DE UN'ENTRATA PRIMA DI HL
01FC   2A4F02                LD     HL, (ENTLEN)
01FF   19                    ADD    HL, DE
0200   EB                    EX     DE, HL
0201   ED4B4F02  MOVEM       LD     BC, (ENTLEN)     ;FA SCORRERE LA MEMORIA IN AVANTI DI UN'ENTRATA
0205   EDB8                  LDDR
0207   3D                    DEC    A
0208   C20102                JP     NZ, MOVEM        ;RIPETI SE NECESSARIO
020B   23                    INC    HL               ;HL È DAVANTI AD UNO SPAZIO VUOTO
020C   FDE5      INSERT      PUSH   IY               ;CARICA L'OGGETTO NELLO SPAZIO VUOTO
020E   D1                    POP    DE
020F   EB                    EX     DE, HL
0210   ED4B4F02              LD     BC, (ENTLEN)
0214   EDB0                  LDIR
0216   3A4C02                LD     A, (TABLEN)      ;INCREMENTA LA LUNGHEZZA DELLA TABELLA
0219   3C                    INC    A
021A   324C02                LD     (TABLEN), A
021D   01FFFF                LD     BC, 0FFFFH       ;MOSTRA CIÒ CHE È STATO ESEGUITO
0220   C9        OUT         RET
                 ;
                 ;
                 ;
0221   CD0001    DELETE      CALL   SEARCH           ;ACCETTA L'INDIRIZZO DELL'OGGETTO
0224   14                    INC    D                ;VEDI SE L'OGGETTO È QUI
0225   CA4902                JP     Z, OUTE
0228   ED5B4F02              LD     DE, (ENTLEN)
022C   EB                    EX     DE, HL
022D   19                    ADD    HL, DE           ;DE È LA LOCAZIONE DELL'OGGETTO, HL È
022E   3A4C02                LD     A, (TABLEN)      ;...UN'ENTRATA PRIMA
0231   90                    SUB    B                ;ORA VEDI QUANTE ENTRATE RIMANGONO
0232   CA3F02                JP     Z, DOWNTAB
0235   ED4B4F02  SHIFTIN     LD     BC, (ENTLEN)
0239   EDB0                  LDIR                    ;FA SCORRERE IN AVANTI LA LUNGHEZZA DI UN'ENTRATA
023B   3D                    DEC    A
023C   C23502                JP     NZ, SHIFTIN
023F   3A4C02    DOWNTAB     LD     A, (TABLEN)      ;DECREMENTA LA LUNGHEZZA DELLA TABELLA
0242   3D                    DEC    A
0243   324C02                LD     (TABLEN), A
0246   01FFFF                LD     BC, 0FFFFH       ;MOSTRA QUALE AZIONE È STATA ESEGUITA
0249   C9        OUTE        RET
                 ;
024A   (0000)    ENDED       END

SYMBOL TABLE

ADDEM   01C9   ADDIT   0196   CLOSEN  024A   COMPRE  024B   DELETE  0221
DOWNTA  023F   ENDED   024A   ENTLEN  024F   ENTRY   011E   FOUND   01BC
HISIDE  01ED   INSERT  020C   MOVEM   0201   MULT    01BD   NEW     01D0
NEXTES  0169   NOGOOD  0142   NOTCLO  0163   NOTFOU  01BA   OUT     0220
OUTE    0249   REALCL  01AF   SEARCH  0100   SETUP   01EE   SHIFTI  0235
TABASE  024D   TABLEN  024C   TESTS   0149   TOOHIG  01A5   TOOLOW  0185
```

Figura 9-23: Programma di ricerca binaria (continua)

LISTA COLLEGATA

La lista collegata si suppone contenga, come al solito, i tre caratteri alfanumerici per la label, seguita da uno a 250 byte di dati, seguita da un puntatore a due byte che contiene l'indirizzo d'inizio del prossimo ingresso, ed alla fine seguita da un "marker" (segnale) di un byte. Ogni volta che questo segnale da un byte è posto ad "1", impedirà che la routine d'inserzione sostituisca un nuovo ingresso al posto di quello già esistente.

Inoltre, un direttorio contiene un puntatore al primo ingresso per ogni lettera dell'alfabeto per facilitare il reperimento. Nel programma è supposto le label siano caratteri alfanumerici ASCII. Tutti i puntatori alla fine della lista sono posti ad un valore NIL che è stato scelto uguale alla base della tabella, in quanto questo valore non dovrebbe mai ricorrere all'interno della lista collegata.

I programmi d'inserzione e di cancellazione eseguono le ovvie manipolazioni sui puntatori. Usano il flag INDEX per indicare che un puntatore, che sta puntando un oggetto, proveniva da un precedente ingresso nella lista o da una tabella a direttorio. I programmi corrispondenti sono mostrati nella Figura 9-29.

La struttura dei dati è mostrata nella Figura 9-25.

```
 DM400                                                                  Valore iniziale
0400  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
                                                                        Listing degli
                                                                        oggetti e loro
                                                                        locazioni in
                                                                        memoria
-DM300
0300  53 4F 4E 31 31 31 31 31 31 31 31 31 31 31 00 00 00  SON111111111...
0310  44 41 44 32 32 32 32 32 32 32 32 32 32 32 00 00 00  DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 33 00 00 00  MOM3333333333...
0330  55 4F 43 34 34 34 34 34 34 34 34 34 34 34 00 00 00  UNC4444444444...
0340  41 4F 54 35 35 35 35 35 35 35 35 35 35 35 00 00 00  ANT5555555555...
0350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0360  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0370  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

-SY
Y=0000 310
 0263/0266
                                  Esegui 'INSERT'
F: 0266 0267
                                                                        Tabella dopo
                                                                        l'inserzione
 DM400
0400  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 33 00 00 00  MOM3333333333...
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

-SY
Y=0320 310
 0263/0266
                                  Esegui 'INSERT' di un altro oggetto
F: 0266 0267
```

Figura 9-24: Lista alfabetica

Listing della tabella dopo l'inserzione. Nota: la tabella è in ordine alfabetico

```
-DM400
0400  44 41 44 32 32 32 32 32-32 32 32 32 32 32 4D 4F 4D  DAD22222222MOM
0410  33 33 33 33 33 33 33 33-33 33 00 00 00 00 00 00 00  3333333333......
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00  ................
```

• • • (inserzioni addizionali) • • •

Configurazione della tabella dopo che tutti gli oggetti sono stati inseriti

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 35 44 41 44  ANT55555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SONL11111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
-SY
Y=0340 300
 G260/263
P=0263 0263'
```

Esegui 'SEARCH' di "SON" (all'indirizzo 0300)

Found

```
-DR
Z  N   A=4E BC=0401 DE=000D HL=0427 S=0100 P=0263 0263' CALL 01D0
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (01D0')
```

Indirizzo dell'oggetto in tabella

(verifica che nella precedente tabella c'è "SON")

```
-G266/269
P=0269 0269'
```

Esegui 'DELETE' su "SON"

Configurazione della tabella dopo la cancellazione. Nota: UNC è stato trasferito in alto. L'ultima entrata UNC deve essere trascurata

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 35 44 41 44  ANT55555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 55-4E 43 34 34 34 34 34 34  3333333UNC444444
0430  34 34 34 34 55 4E 43 34-34 34 34 34 34 34 34 34  4444UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
-G260/263
P=0263 0263'
```

Tenta di eseguire ancora 'SEARCH' (su "SON")

Non trovato

```
-DR
S  N   A=FE BC=0401 DE=FF0D HL=0427 S=0100 P=0263 0263' CALL 01D0
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (01D0')
-G263/266
P=0266 0266'
```

Re-inserisci l'oggetto ("SON")

Figura 9-24: Lista alfabetica (continua)

Configurazione corrente della tabella. Confronta con quella precedente DELETE

```
DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32 32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34 34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Mostra che l'azione è stata eseguita

```
-DR
   A=05 BC=FFFF DE=0434 HL=030D S=0100 P=0266 0266' CALL 0221
0  00 B'=0000 D'=0000 H'=0000 X=0422 Y=0300 I=00        (0221')
```

Figura 9-24: Lista alfabetica (continua)

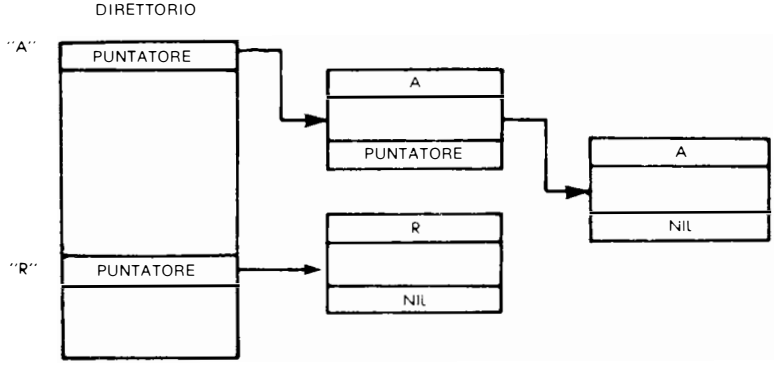
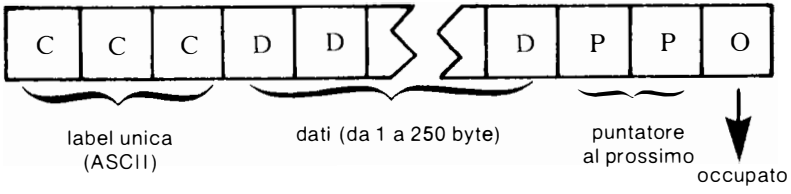


Figura 9-25: Struttura di lista collegata

Una applicazione per questa struttura dei dati potrebbe essere un libro di indirizzi computerizzato, dove ogni persona è rappresentata da un unico codice di tre lettere (forse le comuni iniziali) ed il campo dei dati contiene un indirizzo semplificato, più il numero telefonico (fino a 250 caratteri). Nella Figura 9-23 esaminiamo la struttura più dettagliatamente. Il formato d'ingresso è:



Come al solito le convenzioni sono:

ENTLEN: lunghezza totale dell'elemento (in byte)
TABASE: indirizzo di base di lista

L'indirizzo dell'OBJECT si suppone risieda sempre nel registro IY prima d'entrare nel programma. Qui, REFBASE punta all'indirizzo di base del direttorio, o "tabella del riferimento".

Ogni indirizzo a due byte all'interno di questo direttorio punta al primo evento della lettera al quale corrisponde nella lista. Perciò, ogni gruppo d'ingressi con una prima lettera identica nelle loro label formano veramente una lista separata all'interno dell'intera struttura. Questa caratteristica facilita la ricerca ed è analoga ad un libro d'indirizzi. Notate che nessun dato è mosso durante una inserzione o una cancellazione.

Sono variati solo i puntatori, come in ogni struttura a lista collegata ben progettata.

Se non viene trovato nessun ingresso con una lettera specifica, oppure se non c'è nessun ingresso che segua alfabeticamente uno già esistente, i loro puntatori punteranno all'inizio della tabella ("NIL"). Alla fine della tabella, per convenzione è immagazzinato un valore tale che il valore assoluto della differenza tra esso e "Z" è maggiore della differenza tra "A" e "Z". Questo rappresenta un segnale EOT (Fine della Tabella). Il valore EOT si suppone che occupi qui la stessa quantità di memoria come un ingresso normale; ma se è desiderato potrebbe essere soltanto un byte. Si suppone che le lettere siano lettere alfabetiche nel codice ASCII.

Cambiare questo richiederebbe la variazione della costante nella routine PRETAB.

Il segnale EOT è posto uguale al valore dell'inizio della tabella ("NIL").

Per convenzione, i "puntatori NIL", trovati alla fine di una stringa, o all'interno di una locazione di direttorio che non punta ad una stringa, sono posti uguali al valore della base della tabella per fornire una identificazione unica. Potrebbe essere usata un'altra convenzione. In particolare, un marker diverso per l'EOT porta ad un certo risparmio di tempo, poichè nessun ingresso NIL ha bisogno di essere conservato per ingressi non esistenti.

L'inserzione e la cancellazione sono compiute nel solito modo (vedere la Parte I di questo capitolo) modificando semplicemente i puntatori richiesti. Il flag INDEXED è usato per indicare se il puntatore all'oggetto è nella tabella di riferimento oppure in un altro elemento della stringa.

Ricerca

Il programma SEARCH risiede alle posizioni di memoria da 0100 a 0155 e usa la subroutine PRETAB all'indirizzo 01D2.

Il principio della ricerca è facile:

1 — Prendere l'ingresso del direttorio che corrisponde alla lettera dell'alfabeto nella prima posizione della label dell'OBJECT.
2 — Prendere il puntatore. Far accedere l'elemento. Se è NIL, l'ingresso non esiste.
3 — Se non è NIL, porre l'elemento di fronte all'OBJECT. La ricerca è riuscita se è trovato un adattamento. Se non è trovato, portare il puntatore al prossimo ingresso in basso della lista.
4 — Ritornare al 2.

Un esempio è mostrato nella Figura 9-26.

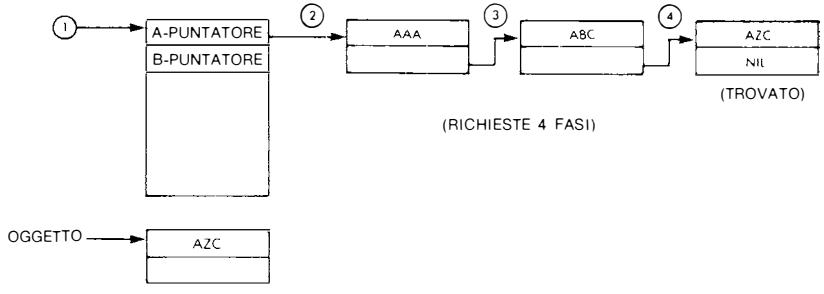


Figura 9-26: Lista collegata — Una ricerca

Inserzione

L'inserzione è essenzialmente una ricerca seguita da un'inserzione una volta che è stato trovato "NIL".

Un blocco di memoria per il nuovo ingresso è collocato dopo il segnale EOT cercando un segnale di occupazione posto a "disponibile".

Il programma è chiamato "NEW" nella Figura 9-29 e risiede agli indirizzi da 0156 ad 1A3. Un esempio è mostrato nella Figura 9-27.

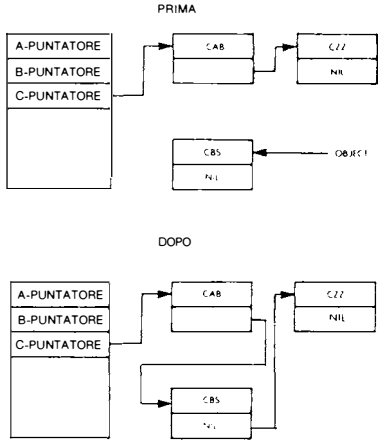


Figura 9-27: Lista collegata: Esempio d'inserzione

Cancellazione

L'elemento è cancellato ponendo il suo segnale di occupazione a "disponibile" e regolando il puntatore ad esso dal direttorio o altrimenti dal precedente elemento.

Il programma è chiamato "DELETE", e risiede agli indirizzi da 01A4 a 01D1.

Nella Figura 9-28 è mostrato un esempio di una cancellazione.

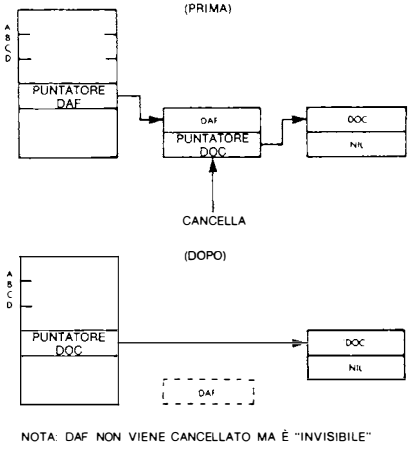


Figura 9-28: Esempio di cancellazione (Lista collegata)

```
0000                        ORG    0100H
     (01E7)   INDEXED       DL     ENDER
     (01E8)   TABASE        DL     ENDER + 1
     (01EA)   REFBASE       DL     ENDER + 3
     (01EC)   ENTLEN        DL     ENDER + 5
              :
0100 3E00     SEARCH       LD     A,0              ;INIZIALIZZA I FLAG
0102 47                    LD     B,A
0103 3C                    INC    A
0104 32E701                LD     (INDEXED),A
0107 CDD201                CALL   PRETAB           ;ACCETTA L'INDIRIZZO DEL PUNTATORE INDICE
010A 1A                    LD     A,(DE)           ;TRASFERISCI I CONTENUTI DEL PUNTATORE IN HL
010B 6F                    LD     L,A
010C 13                    INC    DE
010D 1A                    LD     A,(DE)
010E 67                    LD     H,A
010F E5                    PUSH   HL
0110 DDE1                  POP    IX
0112 DD7E00   COMPARE      LD     A,(IX + 0)       ;OSSERVA LA PRIMA LETTERA DELL'ENTRATA
0115 FE7C                  CP     7CH              ;VEDI SE È IL SEGNALE EOT
0117 D25501                JP     NC,NOTFOUND
011A DD7E00                LD     A, (IX + 0)      ;CONFRONTA LE PRIME LETTERE
011D FDBE00                CP     (IY + 0)
0120 DA3E01                JP     C,NOGOOD
0123 C25501                JP     NZ,NOTFOUND
0126 DD7E01                LD     A, (IX + 1)      ;CONFRONTA LE SECONDE LETTERE
0129 FDBE01                CP     (IY + 1)
012C DA3E01                JP     C,NOGOOD
012F C25501                JP     NZ,NOTFOUND
0132 DD7E02                LD     A, (IX + 2)      ;CONFRONTA LE TERZE LETTERE
0135 FDBE02                CP     (IY + 2)
0138 CA5301                JP     Z,FOUND
013B D25501                JP     NC,NOTFOUND
013E DDE5     NOGOOD       PUSH   IX
0140 D1                    POP    DE
0141 2AEC01                LD     HL,(ENTLEN)      ;SALTA AL PUNTATORE DELL'ENTRATA
0144 19                    ADD    HL,DE
0145 4E                    LD     C,(HL)           ;PONI IL VALORE DEL PUNTATORE IN BC.
0146 23                    INC    HL
0147 46                    LD     B,(HL)
0148 C5                    PUSH   BC               ;CARICA IX CON IL PUNTATORE
0149 DDE1                  POP    IX
014B 3E00                  LD     A,0
014D 32E701                LD     (INDEXED),A      ;RESET FLAG
0150 C31201                JP     COMPARE
0153 06FF     FOUND        LD     B,0FFH
0155 C9       NOTFOUND     RET
              :
              :
              :
0156 CD0001   NEW          CALL   SEARCH           ;VEDI DOVE DOVREBBE ANDARE L'OGGETTO
0159 04                    INC    B
015A CAA301                JP     Z,OUT
015D D5                    PUSH   DE               ;MEMORIZZA L'INDIRIZZO DELL'ENTRATA PRECEDENTE
015E 2AE801                LD     HL,(TABASE)      ;TROVA SPAZIO IN TABELLA
0161 EB       NEXTONE      EX     DE,HL            ;TRASFERISCI ALLA FINE DELL'ENTRATA SUCCESSIVA
0162 2AEC01                LD     HL,(ENTLEN)
0165 23                    INC    HL               ;SOMMA 3 PER OTTENERE L'EFFETTIVA LUNGHEZZA DELL'ENTRATA
0166 23                    INC    HL
0167 23                    INC    HL
0168 19                    ADD    HL,DE
0169 7E                    LD     A,(HL)
016A 3D                    DEC    A
016B CA6101                JP     Z,NEXTONE        ;SE C'È QUALCOSA, PROVA ANCORA
016E 13                    INC    DE
016F D5                    PUSH   DE               ;SALVA LA POSIZIONE DELLO SPAZIO VUOTO
0170 FDE5                  PUSH   IY               ;TRASFERISCI IY IN HL
```

Figura 9-29: Lista collegata — I programmi

```
0172  E1                POP   HL
0173  ED4BEC01          LD    BC,(ENTLEN)     ;TRASFERISCI L'OGGETTO NELLA TABELLA
0177  EDB0              LDIR
0179  DDE5              PUSH  IX              ;PONI L'INDIRIZZO DELL'ENTRATA DOPO L'OGGETTO
017B  E1                POP   HL              ;... NELLA POSIZIONE DEL PUNTATORE
017C  EB                EX    DE,HL
017D  73                LD    (HL),E
017E  23                INC   HL
017F  72                LD    (HL),D
0180  23                INC   HL
0181  3601              LD    (HL),1          ;POSIZIONA IL SEGNALE DI OCCUPATO
0183  E1                POP   HL              ;ACCETTA L'INDIRIZZO DI QUESTO SPAZIO
0184  3AF701            LD    A,(INDEXED)     ;VEDI QUALE DEI PUNTATORI PRECEDENTI DEVE
0187  3D                DEC   A                 ESSERE IMPOSTATO
0188  CA9801            JP    Z,SETINX
018B  E3                EX    (SP),HL         ;ACCETTA L'INDIRIZZO DELL'ENTRATA PRECEDENTE ALLO
018C  ED5BEC01          LD    DE,(ENTLEN)       OGGETTO & MUOVI ALL'AREA DEL PUNTATORE
0190  19                ADD   HL,DE
0191  D1                POP   DE              ;RECUPERA L'INDIRIZZO DELL'OGGETTO
0192  73                LD    (HL),E          ;PONILO NELLA POSIZIONE DEL PUNTATORE
0193  23                INC   HL
0194  72                LD    (HL),D
0195  C3A001            JP    FINISH
0198  C1        SETINX  POP   BC              ;AZZERA LO STACK
0199  CDD201            CALL  PRETAB          ;ACCETTA L'INDIRIZZO INDICE
019C  EB                EX    DE,HL           ;CARICA HL IN QUEST'ULTIMO
019D  73                LD    (HL),E
019E  23                INC   HL
019F  72                LD    (HL),D
01A0  01FFFF    FINISH  LD    BC,0FFFFH       ;MOSTRA CHE È STATO ESEGUITO
01A3  C9        OUT     RET


                ;
01A4  CD0001    DELETE  CALL  SEARCH          ;ACCETTA L'INDIRIZZO DELL'OGGETTO
01A7  04                INC   B               ;VEDI SE È QUI
01A8  C2D101            JP    NZ,OUTE
01AB  DDE5              PUSH  IX              ;PONI HL PER PUNTARE ALL'AREA DELL'OGGETTO
01AD  E1                POP   HL
01AE  ED4BEC01          LD    BC,(ENTLEN)
01B2  09                ADD   HL,BC
01B3  4E                LD    C,(HL)          ;RECUPERA IL PUNTATORE
01B4  23                INC   HL
01B5  46                LD    B,(HL)
01B6  23                INC   HL
01B7  3600              LD    (HL),0          ;RIMUOVI IL SEGNALE DI OCCUPATO
01B9  3AE701            LD    A,(INDEXED)     ;VEDI SE L'INDICE DEVE ESSERE VARIATO
01BC  3D                DEC   A
01BD  C2C701            JP    NZ,CHANGEM
01C0  CDD201            CALL  PRETAB          ;SÌ PONI L'INDIRIZZO IN HL
01C3  EB                EX    DE,HL
01C4  C3CB01            JP    MOVIN
01C7  2AEC01    CHANGEM LD    HL,(ENTLEN)     ;PONI HL AL PUNTATORE DEL PRECEDENTE
01CA  19                ADD   HL,DE
01CB  71        MOVIN   LD    (HL),C          ;PONI L'INDIRIZZO DEL SUCCESSIVO IN UNO QUALSIASI
01CC  23                INC   HL              ;... (INDICE O ENTRATA)
01CD  70                LD    (HL),B
01CE  01FFFF            LD    BC,0FFFFH
01D1  C9        OUTE    RET
                ;
                ;
                ;
01D2  E5        PRETAB  PUSH  HL
01D3  FD7E00            LD    A, (IY + 0)     ;ACCETTA LA PRIMA LETTERA DELL'OGGETTO
01D6  3D                DEC   A               ;RIMUOVI L'ASCII INIZIALE
01D7  D640              SUB   40H
01D9  CB27              SLA   A               ;MOLTIPLICA PER 2
```

Figura 9-29: Lista collegata — I programmi (continua)

```
01DB 2AEA01                LD     HL,(REFBASE)
01DE 85                    ADD    L
01DF 6F                    LD     L,A
01E0 D2E401                JP     NC,FIXUP
01E3 24                    INC    H
01E4 EB          FIXUP     EX     DE,HL
01E5 E1                    POP    HL
01E6 C9                    RET
                 ;
01E7 (0000)      ENDER     END


SYMBOL TABLE

CHANGE  01C7    COMPAR  0112    DELETE  01A4    ENDER   01E7    ENTLEN  01EC
FINISH  01A0    FIXUP   01E4    FOUND   0153    INDEXE  01E7    MOVIN   01CB
NEW     0156    NEXTON  0161    NOGOOD  013E    NOTFOU  0155    OUT     01A3
OUTE    01D1    PRETAB  01D2    REFBAS  01EA    SEARCH  0100    SETINX  0198
TABASE  01E8
```

Figura 9-29: Lista collegata — I programmi (continua)

Oggetti in memoria

Listing degli oggetti e della loro locazione in memoria

```
DM300
0300  53 4E 41 31 31 31 31 31 31 31 31 31 31 00 00 00  SON111111111...
0310  44 41 44 32 32 32 32 32 32 32 32 32 32 00 00 00  DAD2222222222...
0320  4D 41 4D 33 33 33 33 33 33 33 33 33 33 00 00 00  MOM3333333333...
0330  55 4E 43 34 34 34 34 34 34 34 34 34 34 00 00 00  UNC4444444444...
0340  41 4E 54 35 35 35 35 35 35 35 35 35 35 00 00 00  ANT5555555555...
0350  41 41 41 36 36 36 36 36 36 36 36 36 36 00 00 00  AAA6666666666...
0360  41 50 50 37 37 37 37 37 37 37 37 37 37 00 00 00  APP7777777777...
0370  53 49 44 38 38 38 38 38 38 38 38 38 38 00 00 00  SID8888888888...
```

Carattere EOT nella tabella iniziale

```
DM400
0400  10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figura 9-30: Lista collegata — Esecuzione campione

Direttorio iniziale

```
-DM500
0500  00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04  ................
0510  00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04  ................
0520  00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04  ................
0530  00 04 00 04 00 00 00 00 00 00 00 00 00 00 00 00  ................
0540  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0550  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0560  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0570  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Segnale di occupato

Puntatori

Configurazione della tabella dopo diverse inserzioni

```
DM400
0400  2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  (...............
0410  41 4F 54 35 35 35 35 35 35 35 35 35 35 20 04 01  ANT5555555555 ..
0420  44 41 44 32 32 32 32 32 32 32 32 32 32 00 04 01  DAD2222222222...
0430  41 41 41 36 36 36 36 36 36 36 36 36 36 10 04 01  AAA6666666666...
0440  53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 04 01  SONI111111111...
0450  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 04 01  MOM3333333333...
0460  53 49 44 38 38 38 38 38 38 38 38 38 38 40 04 01  SID8888888888@..
0470  41 5A 50 32 32 32 32 32 32 32 32 32 32 00 04 01  AZP2222222222...
```

```
SY
Y 0360 .316
G226/229
F 0229 .0229'
```

Cancella un ingresso

Solo variazione

```
DM400
0400  2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  (...............
0410  41 4F 54 35 35 35 35 35 35 35 35 35 35 20 04 01  ANT5555555555 ..
0420  44 41 44 32 32 32 32 32 32 32 32 32 32 00 04 00  DAD2222222222...
0430  41 41 41 36 36 36 36 36 36 36 36 36 36 10 04 01  AAA6666666666...
0440  53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 04 01  SONI111111111...
0450  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 04 01  MOM3333333333...
0460  53 49 44 38 38 38 38 38 38 38 38 38 38 40 04 01  SID8888888888@..
0470  41 5A 50 32 32 32 32 32 32 32 32 32 32 00 04 01  AZP2222222222...
```

```
-0220/223
F-0223 .0223'
```

Esegui 'SEARCH' per cancellare l'entrata

Non trovata

```
DK
    N    A 37 BC-00FF DE-0400 HL-0000 S-0100 P-0223 .0223' CALL 0171
         A' 00 B'-0000 D' 0000 H'-0000 X-0400 Y-0310 T-00        (0171')
```

```
SY
Y 0310 .340
G220/223
F 0223 .0223'
```

Esegui "SEARCH" di un'entrata inesistente

Non trovata

```
DK
Z   N    A-54 BC-FF10 DE-0430 HL-043E S-0100 P-0223 .0223' CALL 0171
         A'-00 B' 0000 D' 0000 H'-0000 X-0410 Y 0340 T-00        (0171')
G226/229
F 0229 .0229'
```

Cancella

Indirizzo dell'entrata in tabella

Nota: variazioni nei puntatori

```
DM400
0400  2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  (...............
0410  41 4F 54 35 35 35 35 35 35 35 35 35 35 20 04 00  ANT5555555555 ..
0420  44 41 44 32 32 32 32 32 32 32 32 32 32 00 04 00  DAD2222222222...
0430  41 41 41 36 36 36 36 36 36 36 36 36 36 20 04 01  AAA6666666666 ..
0440  53 4F 4E 31 31 31 31 31 31 31 31 31 31 00 04 01  SONI111111111...
0450  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 04 01  MOM3333333333...
0460  53 49 44 38 38 38 38 38 38 38 38 38 38 40 04 01  SID8888888888@..
0470  41 5A 50 32 32 32 32 32 32 32 32 32 32 00 04 01  AZP2222222222...
```

Figura 9-30: Lista collegata — Esecuzione campione (continua)

SOMMARIO

Il programmatore alle prime armi non ha ancora bisogno di occuparsi dei dettagli della gestione ed esecuzione delle strutture dei dati. Comunque, l'efficiente programmazione di algoritmi non banali richiede una buona comprensione delle strutture dei dati.

Gli esempi presentati in questo capitolo dovrebbero aiutare a raggiungere una tale comprensione e risolvere tutti i problemi comuni incontrati con strutture dati ragionevoli.

CAPITOLO 10

SVILUPPO DEL PROGRAMMA

INTRODUZIONE

Tutti i programmi che abbiamo studiato e sviluppato finora, sono stati sviluppati a mano senza l'aiuto di qualche mezzo software o hardware. Il solo miglioramento sulla codifica binaria diretta è stato l'uso di simboli mnemonici, quelli del linguaggio assembly. Per lo sviluppo software effettivo, è necessario capire la gamma degli aiuti per lo sviluppo hardware e software. Lo scopo di questo capitolo è quello di presentare e valutare questi aiuti.

SCELTE DI PROGRAMMAZIONE FONDAMENTALI

Esistono tre alternative fondamentali: scrivere un programma nel binario o nell'esadecimale, scriverlo in un linguaggio a livello assembly, o scriverlo in un linguaggio di alto livello. Esaminiamo queste alternative.

Codifica Esadecimale

Il programma sarà normalmente scritto usando la mnemonica del linguaggio assembly. Comunque, la maggior parte dei sistemi computer su singola scheda ed a basso costo, non forniscono un assembler (assemblatore). L'assembler è il programma che tradurrà automaticamente la mnemonica usata per il programma nei codici binari richiesti. Quando non è disponibile nessun assembler, questa traduzione dalla mnemonica al binario deve essere eseguita manualmente.

Il binario è *spiacevole* da usare e conduce all'errore, cosicchè è normalmente usato l'esadecimale. Nel capitolo 1 è stato mostrato che un digit esadecimale rappresenta quattro bit binari. Perciò, saranno usati due digit esadecimali per rappresentare il contenuto di ogni byte.

Come esempio, nell'Appendice compare la tabella che mostra l'equivalente esadecimale delle istruzioni per lo Z80.

In breve, ogni volta che le risorse del programmatore sono limitate e non è disponibile nessun assembler si dovrà tradurre a mano il programma in esadecimale. Questo può essere fatto ragionevolmente per un piccolo numero di istruzioni, come, forse, da 10 a 100. Per i programmi più grandi, questo processo è tedioso e conduce all'errore, così si tende a non usarlo. Comunque, quasi tutti i microcomputer su scheda singola richiedono l'ingresso dei programmi nel modo esadecimale. Per limitare il loro costo non sono forniti di un assembler e di una tastiera alfanumerica completa.

Ricapitolando, la codifica esadecimale non è la via desiderabile per introdurre un programma in un computer. È semplicemente un modo economico. Il costo di un assemblatore e della tastiera alfanumerica richiesta è da confrontarsi contro l'aumentato lavoro richiesto per introdurre il programma nella memoria. Comunque, questo non cambia il modo in cui è scritto il programma stesso. *Il programma è ancora scritto nel linguaggio a livello assembly* in modo che può essere esaminato dal programmatore umano ed essere significativo.

La Programmazione in Linguaggio Assembly

La programmazione a livello assembly copre sia i programmi che possono essere caricati in esadecimale che quelli che possono essere caricati nel sistema sotto forma simbolica di livello assembly. Esaminiamo adesso l'ingresso di un programma direttamente nella sua rappresentazione in linguaggio assembly. Deve essere disponibile un programma assembler.

L'assembler leggerà tutte le istruzioni mnemoniche del programma e le tradurrà nello schema di bit richiesto usando da 1 a 5 byte, come specificato dalla codifica delle istruzioni. Inoltre, un buon assembler offrirà un numero di possibilità addizionali per scrivere il programma. Queste saranno esaminate più avanti se nella parte sugli assembler. In particolare sono disponibili delle *direttive* che modificheranno il valore dei simboli. Può essere usato l'indirizzamento simbolico e può essere specificata una ramificazione ad una posizione simbolica.

Durante la fase di debugging (messa a punto), quando il programmatore può rimuovere o aggiungere istruzioni, non sarà necessario riscrivere l'intero programma se è inserita una istruzione extra tra una ramificazione e il punto in cui si ramifica, fino a quando sono usate label simboliche. L'assembler si incaricherà di regolare tutte le label durante il processo di traduzione. Inoltre, un assembler permette al programmatore di mettere a punto il suo programma in forma simbolica. Può essere usato un disassembler per esaminare il contenuto di una locazione di memoria e ricostruire l'istruzione a livello assembly che essa rappresenta. Più avanti saranno esaminate le varie risorse software normalmente disponibili su un sistema. Adesso esaminiamo la terza alternativa.

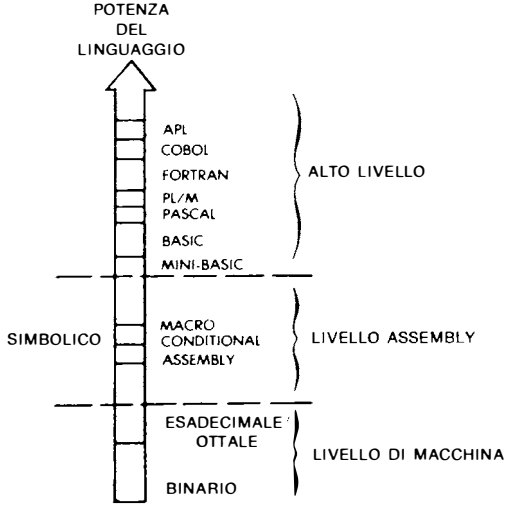


Figura 10-1: Livelli di programmazione

Linguaggio ad alto Livello

Un programma può essere scritto in un linguaggio ad alto livello come il BASIC, APL, PASCL, o altri.

Le tecniche per programmare in questi vari linguaggi sono descritte in libri specifici e non saranno esaminate qui.

Perciò, esamineremo soltanto brevemente questo modo di programmazione. Un linguaggio ad alto livello offre potenti istruzioni le quali rendono più facile e veloce la programmazione. Le istruzioni devono poi essere tradotte tramite un programma complesso nella rappresentazione binaria finale che un microcomputer può eseguire. Tipicamente, ogni istruzione ad alto livello sarà tradotta in un grosso numero di istruzioni binarie individuali. Il programma che esegue questa traduzione automatica è chiamato *compilatore* o *interprete*. Un compilatore tradurrà tutte le istruzioni di un programma in sequenza nel codice oggetto. In una fase separata, il codice che ne risulta sarà poi eseguito. Invece, un interprete interpreterà una istruzione singola, poi l'eseguirà, poi "tradurrà" la successiva per poi eseguirla. Un interprete offre il vantaggio di una risposta interattiva, ma porta ad una efficienza più bassa in confronto al compilatore. Questi soggetti non saranno studiati ulteriormente in questa sede. Ritorniamo alla programmazione di un microprocessore reale nel linguaggio di livello assembly.

IL SUPPORTO SOFTWARE

Esamineremo le principali possibilità di software che sono (o che dovrebbero essere) disponibili nel sistema completo per uno sviluppo software conveniente. Sono già state introdotte alcune delle definizioni. Saranno riassunte qui e il resto dei programmi importanti sarà definito prima di procedere.

L'*assemblatore* o assembler è il programma che traduce la rappresentazione mnemonica delle istruzioni nel loro equivalente binario. Normalmente traduce una istruzione simbolica in una istruzione binaria (la quale può occupare 1, 2, oppure 3 byte). Il codice binario che ne risulta è chiamato *codice oggetto*. È direttamente eseguibile dal microcomputer. Come effetto collaterale, l'assemblatore produrrà anche una lista simbolica completa del programma, cosi come le tabelle di equivalenza che il programmatore deve usare e la lista dei simboli occorrenti nel programma. Gli esempi saranno presentati più avanti in questo capitolo.

Inoltre, l'assemblatore elencherà errori di sintassi come istruzioni sbagliate ortograficamente o illecite, errori di ramificazione, label doppie o label mancanti. Non cancellerà errori *logici* (questo è il vostro problema). Un *compilatore* è il programma che traduce le istruzioni in linguaggio ad alto livello nella loro forma binaria.

Un *interprete* è un programma simile ad un compilatore, che traduce le istruzioni ad alto livello nella loro forma binaria ma non mantiene la rappresentazione intermedia e la esegue immediatamente. Infatti, spesso non genera nemmeno nessun codice intermedio, ma esegue le istruzioni ad alto livello direttamente.

Un *monitor* è il programma fondamentale che è indispensabile per usare le risorse hardware di questo sistema. Controlla continuamente i dispositivi d'input per l'input e dirige il resto dei dispositivi. Come esempio, un monitor minimo per un microcomputer su singola scheda, fornito di una tastiera e di LED, deve esaminare continuamente la tastiera per un input del programmatore e mostrare il contenuto specificato sui diodi emettitori di luce.

Inoltre, deve essere capace di capire un numero di ordini limitati della tastiera, come START, STOP, CONTINUE, LOAD MEMORY, EXAMINE MEMORY.

Su un grande sistema, il monitor è spesso qualificato come il programma *esecutivo*, quando è anche fornito di un complesso file di gestione o una elencazione dei compiti. Il set completo delle possibilità è chiamato *sistema operativo*. Se i file risiedono su un disco, il sistema operativo è qualificato come un *sistema operativo* su disco, o DOS (disk operating system).

Un *editor* è il programma progettato per facilitare l'ingresso e la modificazione del testo o dei programmi. Permette al programmatore di far entrare caratteri in modo conveniente, aggiungerli, inserirli, aggiungere linee, rimuovere linee, cercare caratteri o stringhe. È una risorsa importante per l'efficace e conveniente ingresso del testo.

Un *debugger* (dispositivo per la messa a punto) è un accessorio necessario per mettere a

punto i programmi. Quando un programma non funziona correttamente, non c'è nessuna indicazione della causa. Il programmatore, perciò, desidera inserire un breakpoint (punto di arresto) nel programma per sospendere l'esecuzione del programma ad indirizzi specificati, ed essere in grado di esaminare il contenuto dei registri o della memoria a questo punto. Questa è la funzione primaria di un debugger. Il debugger tiene conto della possibilità di sospendere un programma, riassumere l'esecuzione, esaminare, mostrare e modificare il contenuto dei registri o della memoria. Un buon debugger sarà fornito di un numero di possibilità addizionali, come la capacità di esaminare dati in forma simbolica, esadecimale, binaria, o altre rappresentazioni solite, così come di caricare i dati in questo formato.

Un *caricatore*, o *linking loader*, porrà vari blocchi di codice oggetto a specificate locazioni nella memoria e regolerà i loro rispettivi puntatori simbolici a quello cui si riferiranno l'un l'altro. È usato per riallocare i programmi o i blocchi nelle varie aree di memoria. Un programma *simulatore* o *emulatore* è usato per simulare il funzionamento di un dispositivo, di solito il microprocessore, in sua assenza, quando si sviluppa un programma su un processore simulato prima di porlo sulla scheda reale. Usando questo approccio, diventa possibile sospendere il programma, modificarlo e mantenerlo nella memoria RAM. Gli svantaggi di un simulatore sono che:

1 – Di solito simula solo il processore stesso, non dispositivi di input output.
2 – La velocità di esecuzione è lenta, e si opera in tempo simulato. Perciò non è possibile provare dispositivi in tempo reale e possono aversi problemi di sincronizzazione anche se può essere trovata corretta la logica del programma.

Un *emulatore* è essenzialmente un simulatore in tempo reale. Usa un processore per simulare un altro, e lo simula nei dettagli completi.

Le *routine di utilità* (utility routines) sono essenzialmente tutte le routine che sono necessarie nella maggior parte delle applicazioni e che il programmatore spera che il produttore abbia fornito!

Possono includere la moltiplicazione, la divisione e altre operazioni aritmetiche, le routine per il movimento dei blocchi, i test dei caratteri, i manipolatori di dispositivo input output (o driver) ed altri.

SEQUENZA DI SVILUPPO DEL PROGRAMMA

Ora esamineremo una successione tipica per sviluppare un programma di livello assembly. Supporremo che siano disponibili tutte le solite possibilità software per dimostrare il loro valore. Se non dovessero essere disponibili in un sistema particolare, sarà tuttavia possibile sviluppare programmi, ma la convenienza sarà diminuita e perciò sarà probabilmente aumentato il tempo necessario per mettere a punto il programma (debugging).

L'approccio normale è quello di progettare un algoritmo e definire le strutture dei dati per il problema che deve essere risolto. Poi, è sviluppato un insieme completo di diagrammi di flusso che rappresenta il flusso del programma. Alla fine, i diagrammi di flusso sono tradotti nel linguaggio di livello assembly per il microprocessore; questa è la fase di codifica.

Successivamente, il programma deve essere caricato nel computer. Nella prossima parte esamineremo le opzioni hardware che devono essere usate in questa fase.

Il programma è fatto entrare nella memoria RAM del sistema sotto il controllo dell'editor. Una volta che è fatta entrare una sezione del programma, come una o più subroutine, essa sarà provata.

Per prima cosa sarà usato l'assemblatore. Se l'assemblatore non risiede già nel sistema, viene caricato da una memoria esterna, come un disco. Poi il programma sarà assemblato,

cioè, tradotto in un codice binario. Questo porta alla realizzazione del programma oggetto, pronto per essere eseguito.

Normalmente non ci si aspetta che un programma funzioni correttamente la prima volta. Per verificare il progetto funzionante, saranno normalmente posti un numero di punti di arresto (breakpoint) nelle posizioni cruciali dove è facile controllare se i risultati intermedi sono corretti. Il debugger sarà usato per questo scopo. I breakpoint saranno specificati alle posizioni scelte. Poi sarà emesso un comando "GO" cosicchè inizia l'esecuzione del programma. Il programma si fermerà automaticamente ad ognuno dei breakpoint specificati.

Il programmatore può allora verificare, esaminando il contenuto dei registri, o della memoria, che fino a quel punto i dati siano corretti. Se sono corretti, si procede fino al prossimo breakpoint. Ogni volta che troviamo dei dati non corretti, è stato rivelato un errore nel programma. A questo punto, il programmatore normalmente si riferisce alla sua lista del programma e verifica se la sua codifica è stata corretta. Se non è trovato nessun errore nella programmazione, l'errore potrebbe essere un errore logico e ci si potrebbe riferire al diagramma di flusso. Qui supporremo che i diagrammi di flusso siano stati controllati normalmente e si presume siano ragionevolmente corretti. È probabile che l'errore venga dalla codifica. Perciò, sarà necessario modificare una sezione del programma. Se la rappresentazione simbolica del programma è ancora nella memoria, noi semplicemente registreremo ancora l'editor e modificheremo le linee richieste, e poi ripeteremo di nuovo la successione precedente. In alcuni sistemi, la memoria disponibile può non essere abbastanza grande, cosi che è necessario liberare la rappresentazione simbolica del programma su un disco o cassetta prima di eseguire il codice oggetto. Naturalmente, in tale caso, si dovrebbe caricare la rappresentazione simbolica del programma dal suo mezzo di supporto prima di far entrare di nuovo l'editor.

Questa procedura sarà ripetuta fino a che sono corretti i risultati del programma.

Sottolineiamo che la prevenzione è molto più efficace della cura. Un progetto corretto porterà tipicamente in un programma che funziona correttamente subito dopo che sono stati rimossi i soliti errori di battitura o gli ovvi errori di codifica. Comunque, progetti disordinati possono portare a programmi che richiederanno un tempo estremamente lungo per essere messi a punto. Il tempo di debugging è generalmente considerato essere molto più lungo del vero tempo di progettazione. In breve, è sempre conveniente impiegare più tempo nel progetto per accorciare la fase di debugging. Comunque, usando questo approccio, è possibile provare l'organizzazione globale del programma, ma non provarlo in tempo reale con dispositivi input/output. Se devono essere provati i dispositivi input/output, la soluzione immediata consiste nel trasferimento del programma su EPROM, installarla sulla scheda e poi vedere se funziona.

C'è una soluzione migliore. È l'uso di un *emulatore* in circuito (in-circuit emulator). Un emulatore in circuito usa il microprocessore Z80 (o qualsiasi altro) per emulare uno Z80 in tempo quasi reale. Emula lo Z80 fisicamente. L'emulatore è fornito di un cavo che termina con un connettore da 40 pin, esattamente identico al pin-out di uno Z80. Questo connettore può essere poi inserito sulla scheda reale di applicazione che si sta sviluppando. I segnali generati dall'emulatore saranno esattamente quelli dello Z80, forse soltanto un po' più lenti. Il vantaggio sostanziale è quello che il programma sotto test risiederà ancora nella memoria RAM del sistema di sviluppo. Genererà i segnali reali che comunicheranno con i dispositivi reali di input/output che si desidera usare. Come risultato, diventa possibile far continuare a sviluppare il programma usando tutte le risorse del sistema di sviluppo (editor, debugger, prestazioni simboliche, sistema di file) mentre si prova l'input/output in tempo reale.

Inoltre, un buon emulatore, fornirà possibilità speciali come un *trace* (traccia). Un trace è una registrazione delle ultime istruzioni o stati di vari bus dei dati nel sistema prima di un breakpoint. In breve, un trace fornisce il film degli eventi che sono avvenuti prima del breakpoint o del malfunzionamento.

Può anche far scattare uno scope ad un indirizzo specificato o al verificarsi di una specificata combinazione di bit. Tale possibilità è di grande valore, poichè quando è trovato un errore è di solito troppo tardi. L'istruzione, o i dati, che hanno causato l'errore è avvenuto prima della rivelazione. La disponibilità di un trace permette al programmatore di trovare quale segmento del programma ha causato l'errore.

Se il trace non è abbastanza lungo, noi porremo semplicemente un breakpoint precedente.

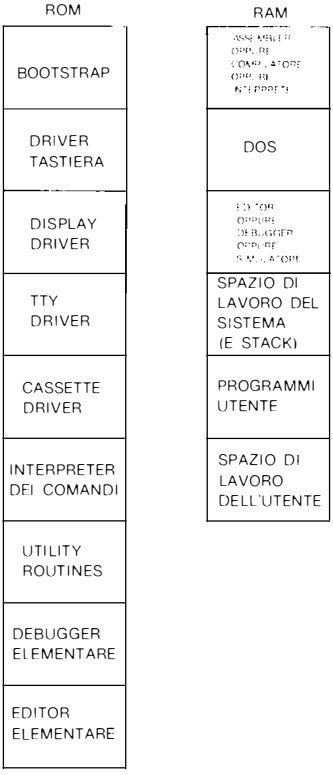


Figura 10-2: Una tipica mappa di memoria

Questo completa la nostra descrizione della solita sequenza di eventi implicati nello sviluppo di un programma. Adesso esaminiamo le alternative hardware disponibili per sviluppare programmi.

ALTERNATIVE HARDWARE

Microcomputer su Singola Scheda

Il microcomputer su singola scheda offre l'approccio a costo più basso allo sviluppo dei programmi. È normalmente fornito di una tastiera esadecimale, più alcuni tasti di funzione, più 6 LED che possono mostrare l'indirizzo ed i dati.

Un assemblatore non è di solito disponibile, poichè il sistema è fornito di un una piccola quantità di memoria.

Al massimo, ha un piccolo monitor e virtualmente nessuna possibilità di editing o di debugging, eccetto che per alcuni comandi. Perciò, tutti i programmi devono essere fatti entrare nella forma esadecimale. Saranno anche mostrati nella forma esadecimale sui LED. In teoria, un microcomputer su singola scheda ha la stessa potenza hardware di ogni altro computer. Sem-

plicemente a causa della sua ristretta dimensione della memoria e della tastiera, non sostiene tutte le solite possibilità di un sistema più grande e rende più lungo lo sviluppo dei programmi. Siccome è tedioso sviluppare programmi nel formato esadecimale, un microcomputer su singola scheda è più adatto per scopi didattici e di addestramento dove devono essere sviluppati programmi di lunghezza limitata e la loro breve lunghezza non è un ostacolo alla programmazione. Comunque, non possono essere usati per lo sviluppo di programmi complessi a meno che non siano aggiunte addizionali schede di memoria e non siano resi disponibili i soliti aiuti software.

Il Sistema di Sviluppo

Un sistema di sviluppo è un sistema microcomputer fornito di una significativa quantità di memoria RAM (32 K, 48 K), con dispositivi di input output richiesti, come un display, una stampante, disk, e di solito, un programmatore di PROM, come pure, forse, di un emulatore in circuito.

Un sistema di sviluppo è specificatamente progettato per facilitare lo sviluppo dei programmi in un ambiente industriale. Normalmente offre tutte, o la maggior parte, delle possibilità software che abbiamo menzionato nel paragrafo precedente. In teoria, è lo strumento dello sviluppo software ideale. La limitazione di un sistema di sviluppo per microcomputer è quello che può non essere capace di sostenere un compilatore o un interprete. Questo perchè un compilatore richiede tipicamente una quantità molto grande di memoria, spesso di più di quanto sia disponibile sul sistema. Comunque, per sviluppare programmi in linguaggio di livello assembly, offre tutte le possibilità richieste.

Poichè i sistemi di sviluppo si vendono in numero relativamente piccolo rispetto ai computer per hobby il loro costo è significativamente più alto.

Microcomputer di Tipo Hobby

L'hardware di un microcomputer di tipo hobby, naturalmente, è esattamente analogo a quello di un sistema di sviluppo. La differenza principale sta nel fatto che normalmente non è fornito dei sofisticati aiuti di sviluppo software, i quali sono disponibili su un sistema di sviluppo industriale. Come esempio, molti microcomputer di tipo hobby offrono solo assemblatori elementari, editor minimi, sistemi di file minimo, nessuna possibilità di connettere un programmatore di PROM, nessun emulatore in circuito, nessun debugger potente. Perciò, rappresentano una fase intermedia tra il microcomputer su singola scheda ed il sistema di sviluppo del microprocessore completo. Per un programmatore che desidera sviluppare programmi di modesta complessità, questi sono probabilmente il miglior compromesso, dal momento in cui offrono il vantaggio del basso costo e un ordine ragionevole di strumenti di sviluppo software, anche se sono abbastanza limitati per la loro convenienza.

Sistema a Divisione di Tempo (Time - Sharing)

È possibile prendere in affitto terminali da varie compagnie per il collegamento a reti time-sharing.

Questi terminali dividono il tempo del computer più grande e beneficiano di tutti i vantaggi delle grandi installazioni.

Gli *assemblatori incrociati* sono disponibili per tutti i microcomputer virtualmente su tutti i sistemi a time - sharing.

Un assemblatore incrociato è semplicemente un assemblatore, diciamo, per uno Z80 il quale risiede, per esempio, in un IBM 370. Formalmente, un assemblatore incrociato è un assemblatore per il microprocessore X, che risiede sul processore Y. La natura del computer che si sta usando è irrilevante. Il programmatore scrive un programma nel linguaggio a livello di assemblaggio dello Z80 e l'assemblatore incrociato lo traduce nell'esatto schema binario. Co-

munque, la differenza è che il programma non può essere eseguito a questo punto. Può essere eseguito da un processore simulato, se ne è disponibile uno, purchè non usi nessuna risorsa di input/output. Perciò, questa soluzione è usata solo in ambienti industriali.

In-House Computer

Ogni volta che è disponibile un grande in-house computer, possono anche essere disponibili assemblatori incrociati per facilitare lo sviluppo dei programmi. Se tale computer offre servizio time-shared, questa scelta è essenzialmente analoga a quella precedente. Se offre solo servizio batch (collettivo), questo è probabilmente uno dei metodi meno convenienti di sviluppo dei programmi, poichè sottomettere i programmi nel modo batch al livello assembly per un microprocessore porta a un tempo di sviluppo molto lungo.

Pannello Frontale o Assenza di Pannello Frontale?

Il pannello frontale è un accessorio hardware spesso usato per facilitare il debugging (messa a punto) dei programmi.

È stato tradizionalmente uno strumento per mostrare convenientemente il contenuto binario di un registro o di una locazione di memoria. Comunque, tutte le funzioni di un pannello di controllo possono essere compiute da un terminale, e il predominio dei display CRT offre ora un servizio quasi equivalente al pannello di controllo mostrando il valore binario dei bit. Il vantaggio supplementare dell'uso del display CRT è quello che si può mutare a piacimento dalla rappresentazione binaria all'esadecimale, al simbolico, al decimale (se sono disponibili le approppriate routine di conversione, naturalmente). Lo svantaggio del CRT è che si devono battere diversi tasti per ottenere il display approppriato invece che una manopola. Comunque, poichè il costo per procurarsi un pannello di controllo è abbastanza sostanziale, i microprocessori più recenti hanno abbandonato questo strumento di debugging. Il valore del pannello di controllo è spesso considerato più sulla base di argomenti emozionali influenzati dalle proprie esperienze passate piuttosto che dall'uso della ragione. Non è indispensabile.

Sommario delle Risorse Hardware

Possono essere distinti tre casi distinti. Se avete soltanto un budget minimo e se desiderate imparare come programmare, comprate un microcomputer su singola scheda. Usandolo, sarete in grado di sviluppare tutti i programmi semplici in questo libro e molti di più.

Comunque, alla fine quando volete sviluppare programmi di oltre alcune centinaia di istruzioni, sentirete la limitazione di questo approccio.

Se siete un programmatore industriale, avrete bisogno di un sistema di sviluppo completo. Qualsiasi soluzione inferiore del sistema di sviluppo completo provocherà un tempo di sviluppo significativamente più lungo.

Il compromesso è chiaro: risorse hardware contro il tempo di programmazione. Naturalmente, se i programmi che devono essere sviluppati sono abbastanza semplici, può essere usato un approccio meno costoso.

Comunque, è difficile giustificare qualsiasi economia hardware quando si compra un sistema di sviluppo, poichè i costi di programmazione saranno di gran lunga il costo dominante del progetto.

Per un computerista personale, un microcomputer di tipo hobby offre possibilità sufficienti anche se minime.

Un buon software di sviluppo per molti dei computer da hobby non è ancora disponibile. Il programmatore dovrà valutare il suo sistema in vista dei commenti presentati in questo capitolo.

Adesso analizziamo dettagliatamente la risorsa più indispensabile: l'assemblatore.

L'ASSEMBLATORE

Durante tutto questo libro abbiamo usato il linguaggio di livello assembly senza presentare la sintassi formale o la definizione del linguaggio di livello assembly. È arrivato il momento di presentare questa definizione. Un assemblatore è progettato per permettere la conveniente rappresentazione simbolica del programma di colui che lo usa e di rendere semplice al programma assemblatore la conversione di questa mnemonica rappresentazione binaria.

Campi dell'Assemblatore

Quando battete un programma per l'assemblatore, abbiamo visto che sono usati i campi. Essi sono:

Il *campo della label* (label field), opzionale, che può contenere un indirizzo simbolico per le istruzioni che seguono.

Il *campo dell'istruzione* (instruction field), che include il codice operativo e qualsiasi operando (può essere distinto un campo dell'operando separato).

Il *campo del commento* (comment field), sulla destra, il quale è opzionale ed è progettato per chiarire il programma.

Questi campi sono mostrati nel modulo di programmazione in Figura 10-3.

Una volta che il programma è stato fornito all'assemblatore, l'assemblatore ne produrrà una *lista*. Quando si genera una lista l'assemblatore fornirà tre campi supplementari, di solito alla sinistra della pagina. Nella Figura 10-4 compare un esempio. Nella parte più a sinistra c'è il numero di riga. Ad ogni riga che è stata battuta dal programmatore è assegnato un numero di riga simbolico.

Il campo seguente alla destra è il campo d'indirizzo, il quale mostra nell'esadecimale il valore del PC che punterà a quell'istruzione.

Muovendo ancora più a destra, troviamo la rappresentazione esadecimale dell'istruzione.

Questo mostra uno dei possibili usi di un assemblatore. Anche se stiamo progettando programmi per microcomputer su singola scheda che accetta solo l'esadecimale, noi dovremmo tuttavia scrivere il programma nel linguaggio di livello assembly, purchè abbiamo accesso ad un sistema fornito di un assemblatore. Poi possiamo mandarlo in esecuzione sul sistema, usando un assemblatore. L'assemblatore genererà automaticamente i codici esadecimali corretti sul nostro sistema. Questo mostra, in un semplice esempio, il valore delle risorse software supplementari.

Tabelle

Quando un assemblatore traduce il programma simbolico nella sua rappresentazione simbolica, compie due compiti essenziali:

1 - Traduce le istruzioni mnemoniche nella loro codifica binaria.

2 Traduce nella loro rappresentazione binaria i simboli usati per le costanti e per gli indirizzi.

Per facilitare il debugging dei programmi, l'assemblatore alla fine della lista mostra l'equivalenza tra il simbolo usato e il suo valore esadecimale.

Questa è chiamata la tabella dei simboli (symbol table).

Alcune tabelle dei simboli non solo elencheranno il simbolo e il suo valore, ma anche i numeri di riga dove compaiono i simboli, fornendo perciò una possibilità supplementare.

Messaggi di Errore

Durante il processo di assemblaggio, l'assemblatore rivelerà gli errori di sintassi e li includerà come parte della lista finale. I diagnostici tipici includono: simboli indefiniti, label già definita, codice operativo non consentito, indirizzo non consentito, modo d'indirizzamento non consentito. Sono naturalmente desiderabili e sono di solito forniti molti diagnostici più dettagliati. Variano con il tipo di assemblatore.

ISTRUZIONE ESADECIMALE					SIMBOLICO			COMMENTI
INDI-RIZZO	1	2	3	4	LABEL	CODICE OPERATIVO	OPERAND	

Figura 10-3: Forma di programmazione di microprocessore

Il Linguaggio Assembly

I codici operativi sono già stati definiti. Qui definiremo i simboli, le costanti e gli operatori che possono essere usati come parte della sintassi dell'assemblatore.

Simboli

I simboli sono usati per rappresentare valori numerici, dati o indirizzi. I simboli possono includere fino a sei caratteri, e devono iniziare con un carattere alfabetico.

I caratteri sono limitati alle lettere dell'alfabeto ed ai numeri. Inoltre, il programmatore non può scegliere nomi identici ai codici operativi utilizzati dallo Z80, di nomi di registri come A, B, C, D, E, H, L, BC, DE, HL, AF, IX, IY, SP così come ai vari nomi abbreviati usati dall'assemblatore come pseudo-operatori. I nomi di queste direttive d'assemblatore sono elencate ai paragrafi corrispondenti. Inoltre, le abbreviazioni usate per designare i flag non dovrebbero essere usate come simboli: C, Z, N, PE, NC, P, PO.

Assegnazione di un Valore ad un Simbolo

Le label sono simboli speciali i cui valori non hanno bisogno di essere definiti dal programmatore. Il valore sarà definito automaticamente dal programma assemblatore ogni volta che si trova quella label. Perciò il valore della label corrisponde automaticamente al numero della linea dove compare.

Sono disponibili pseudo-istruzioni speciali per forzare un nuovo valore di inizio per le label, o per assegnare loro un valore specifico.

```
  CROMEMCO CDOS Z80 ASSEMBLER version 02. 15          PAGE 0001
0000'            0001          ORG     0100H
     (0200)      0002  MPRAD   DL      0200H
     (0202)      0003  MPDAD   DL      0202H
     (0204)      0004  RESAD   DL      0204H
                 0005  ;
0100 ED4B0002    0006  MP488   LD      BC, (MPRAD)   ;CARICA IL MOLTIPLICATORE IN C
0104 0608        0007          LD      B8            ;B È IL CONTATORE DI BIT
0106 ED5B0202    0008          LD      DE, (MPDAD)   ;CARICA IL MOLTIPLICANDO IN E
010A 1600        0009          LD      D, 0          ;AZZERA D
010C 210000      0010          LD      HL, 0         ;PONI A ZERO IL RISULTATO
010F CB39        0011  MULT    SRL     C             ;FA SCORRERE IL BIT DEL MOLTIPLICATORE NEL RIPORTO
0111 3001        0012          JR      NC, NOADD     ;PROVA IL RIPORTO
0113 19          0013          ADD     HL, DE        ;SOMMA MPD AL RISULTATO
0114 CB23        0014  NOADD   SLA     E             ;FA SCORRERE MPD A SINISTRA
0116 CB12        0015          RL      D             ;SALVA IL BIT IN D
0118 05          0016          DEC     B             ;DECREMENTA IL CONTATORE DI SCORRIMENTI
0119 C20F01      0017          JP      NZ, MULT      ;RIPETI SE CONTATORE < >0
011C 220402      0018          LD      (RESAD), HL   ;MEMORIZZA IL RISULTATO
011F (0000)      0019          END

Errors           0
```

Figura 10-4: Output dell'assemblatore — Un esempio

Comunque, altri simboli usati per le costanti o per gli indirizzi di memoria devono essere definiti dal programmatore prima del loro uso.

Per assegnare un valore ad ogni simbolo può essere usato una *direttiva* d'assemblatore speciale. Una *direttiva* è essenzialmente una istruzione all'assemblatore che non sarà tradotta in una situazione eseguibile. Per esempio, la costante LOG sarà definita come:

```
LOG   DFW   3002H
```

Questo assegna il valore esadecimale 3002 alla variabile LOG.

Le direttive d'assemblatore saranno esaminate in dettaglio in un paragrafo successivo.

Costanti o Literal

Le costanti possono essere espresse tradizionalmente in decimale, in esadecimale, in ottale, oppure in binario, oppure come stringhe alfanumeriche. Deve essere usato un simbolo per differenziare le varie basi usate per rappresentare il numero. Per caricare "0" nell'accumulatore, scriveremo semplicemente:

```
LD  A, 0
```

Opzionalmente può essere usato un "D" alla fine della costante. Un numero esadecimale sarà terminato dal simbolo "H". Per caricare il valore "FF" nell'accumulatore, scriveremo:

```
LD  A, FFH
```

Un simbolo ottale è terminato dal simbolo "0" o "Q". Un simbolo binario è terminato da "B".
Per esempio, per caricare il valore "11111111" nell'accumulatore, scriveremo:

```
LD  A,  11111111B
```

I caratteri literal ASCII possono essere usati anche nel campo literal. Il simbolo ASCII deve essere incluso in virgolette singole.
Per esempio, per caricare il simbolo "S" nell'accumulatore, scriveremo:

```
LD  A,  'S'
```

Esercizio 10-1: *Le seguenti due istruzioni caricheranno lo stesso valore nell'accumulatore: LD A, '5'. e LD A, 5 H?*

Notate che nella convenzione della Zilog, le parentesi denotano un indirizzo. Per esempio:

```
LD   A,  (10)
```

specifica che l'accumulatore è caricato dal contenuto della posizione di memoria 10 (decimale).

Operatori

Per facilitare ulteriormente la scrittura di programmi simbolici, gli assemblatori permettono l'uso di operatori. Al minimo, dovrebbero permettere più o meno in modo che si possa specificare, per esempio:

```
LD  A,  (ADDRESS)
LD  A,  (ADDRESS + 1)
```

È importante capire che l'espressione ADDRESS + 1 sarà calcolata dall'assemblatore per determinare il vero indirizzo di memoria che deve essere inserito come l'equivalente binario. Sarà calcolato nel *tempo di assembly* e non durante l'esecuzione del programma.
Inoltre, possono essere disponibili altri operatori, come la moltiplicazione e la divisione, una convenienza quando si accede a tabelle nella memoria. Possono anche essere disponibili operatori più specializzati, come maggiore di o minore di, i quali troncano un valore da due byte rispettivamente nel byte alto e basso.
Naturalmente, una espressione deve *valutare* ad un valore positivo. I numeri negativi normalmente possono non essere usati e dovrebbero essere espressi in un formato esadecimale.
Infine, è tradizionalmente usato un simbolo speciale per rappresentare il valore corrente dell'indirizzo della riga: "$". Questo simbolo dovrebbe essere interpretato come "locazione corrente" (valore di PC).

Esercizio 10-2: *Qual è la differenza tra le seguenti istruzioni?*

```
LD  A,  10101010 B
LD  A,  (10101010 B)
```

Esercizio 10-3: *Qual è l'effetto della seguente istruzione?*

```
JP  NC,  $  2
```

Espressioni

Le specifiche dell'assemblatore Z80 permettono una vasta gamma di espressioni con operazioni aritmetiche e logiche.

L'assemblatore valuterà le espressioni da sinistra a destra, usando le priorità specificate dalla tabella nella Figura 10-5. Le parentesi possono essere usate per forzare un ordine specifico di valutazione. Comunque, le parentesi più esterne denoteranno che il contenuto deve essere trattato come un indirizzo.

Direttive dell'Assemblatore

Le direttive sono ordini speciali dati dal programmatore ail'assemblatore, le quali causano l'immagazzinamento di certi valori in simboli o nella memoria, o controllano l'esecuzione o i modi di stampa dell'assemblatore. L'insieme dei comandi che controlla specificamente i modi di stampa dell'assemblatore è anche chiamato "comandi" ed è descritto in un paragrafo separato.

Per fornire un esempio specifico, qui esaminiamo le 11 direttive dell'assemblatore disponibili sul sistema di sviluppo Zilog:

```
ORG  nn
```

Questa direttiva posizionerà il contatore d'indirizzo dell'assemblatore al valore nn. In altre parole, la prima istruzione eseguibile incontrata dopo questa direttiva risiederà al valore nn. Può essere usata per localizzare segmenti differenti di un programma in locazioni di memoria differenti.

```
EQU  nn
```

Questa direttiva è usata per assegnare un valore ad una label.

```
DEFL  nn
```

Anche questa direttiva assegna un valore n ad una label, ma può essere ripetuta all'interno del programma con valori differenti per la stessa label, mentre EQU può essere usata solo una volta.

```
DEFB  n
```

Questa direttiva assegna il contenuto di otto bit ad un byte che risiede al contatore di referimento corrente.

```
DFB  'S'
```

assegna il valore ASCII di "S" al byte.

```
DEFW  nn
```

Questa assegna il valore nn alla parola a due byte che risiede al contatore di referimento corrente nella posizione che segue.

OPERATORE	FUNZIONE	PRIORITÀ
	PIÙ	1
	MENO	1
.NOT. oppure	NOT LOGICO	1
.RES.	RISULTATO	1
**	ELEVAMENTO A POTENZA	2
*	MOLTIPLICAZIONE	3
	DIVISIONE	3
.MOD.	MODULO	3
.SHR.	SCORRIMENTO LOGICO A DESTRA	3
.SHL.	SCORRIMENTO LOGICO A SINISTRA	3
+	ADDIZIONE	4
	SOTTRAZIONE	4
.AND. oppure &	AND LOGICO	5
.OR. oppure |	OR LOGICO	6
.XOR.	XOR LOGICO	6
.EQ. oppure =	UGUALE	7
.GT. oppure >	MAGGIORE DI	7
.LT. oppure <	MINORE DI	7
.UGT.	MAGGIORE DI SENZA SEGNO	7
.ULT.	MINORE DI SENZA SEGNO	7

Figura 10-5: Priorità degli operatori

DEFS nn

riserva un blocco di byte di memoria di dimensione nn che inizia al valore corrente del contatore di referimento.

DEFM 'S'

immagazzina nella memoria la stringa 'S' che inizia al contatore di referimento corrente. Deve essere minore di 63 in lunghezza.

MACRO P0 P1 Pn

è usata per definire una label come una macro, e per definire la sua lista di parametri formali. Le macro sono definite in un altro paragrafo successivo.

END

indica la fine del programma. Sarà ignorata ogni altra dichiarazione che la segue.

ENDM

è usata per segnare la fine di una definizione macro.

Comandi dell'Assemblatore

I *comandi* vengono impiegati per modificare il formato del listing e per controllare i modi di stampa dell'assemblatore. Tutti i comandi iniziano con un asterisco in colonna uno. L'assemblatore dello Z80 fornisce sette comandi. Esempi tipici sono i seguenti:

```
EJECT
```

che impone al listing di passare all'inizio della pagina successiva; e

```
LIST OFF
```

che impone la sospensione della stampa. Gli altri comandi sono: "* HEADING S", "* LIST ON", "* MACLIST ON", "*MACLIST OFF", "* INCLUDE FILENAME".

Macro

Una macro è semplicemente un nome assegnato ad un gruppo di istruzioni. Si tratta di una caratteristica conveniente per il programmatore. Qualora un gruppo di istruzioni venga usato diverse volte in un programma, è possibile definire una macro per rappresentarlo, invece di dover riscrivere ogni volta tale gruppo di istruzioni.

Per esempio, potremmo scrivere:

```
SAVREG    MACRO   PUSH AF
                  PUSH BC
                  PUSH DE
                  PUSH HL
                  ENDM
```

poi scrivere semplicemente il nome "SAVREG" invece delle istruzioni precedenti. Ogni volta che scriviamo SAVREG, le cinque righe corrispondenti saranno sostituite invece del nome. Un assemblatore fornito di una possibilità macro è chiamato macro-assemblatore. Quando l'assemblatore macro incontra un SAVREG, esegue una mera sostituzione fisica delle righe equivalenti.

Macro o Subroutine?

A questo punto, una macro può sembrare che operi in un modo analogo ad una subroutine. Questo non è vero. Quando l'assemblatore è usato per produrre il codice oggetto, ogni volta che è incontrato un nome macro, sarà rimpiazzato dalla vera istruzione di cui sta al posto. Al tempo di esecuzione, il gruppo di istruzioni apparirà tante volte quanto è comparso il nome della macro. Per contrasto, una subroutine è definita solo una volta, e poi può essere usata ripetutamente, il programma salterà all'indirizzo di subroutine. Una macro è eseguita durante il *tempo d'assemblaggio*. Una subroutine è eseguita durante il *tempo d'esecuzione*. Il loro funzionamento è abbastanza differente.

Parametri Macro

Ogni macro può essere fornita di un numero di parametri. Come esempio, consideriamo la seguente macro:

```
SWAP      MACRO   #M, #N, #T
          LD      A, #M          MINA
          LD      #T, A          AINT(M)
```

```
LD      A, #N       N IN A
LD      #M, A       A IN M (=N)
LD      A, #T       T IN A
LD      #N, A       A IN N (=T)
END     M
```

Questa macro effettua uno "swapping" (scambio) tra il contenuto delle locazioni di memoria M e N. Uno scambio tra due registri, o due posizioni di memoria, è una operazione che non è fornita dallo Z80. Può essere usata una macro per eseguirla. "T" in questo caso è semplicemente il nome per una posizione d'immagazzinamento temporaneo richiesto dal programma. Come esempio, scambiamo il contenuto delle locazioni di memoria ALPHA e BETA.

L'istruzione che fa questo è la seguente:

```
SWAP   (ALPHA),   (BETA),   (TEMP)
```

In questa istruzione, TEMP è il nome di qualche posizione di immagazzinamento temporaneo, che noi sappiamo essere disponibile e che può essere usata dalla macro. La risultante espansione della macro è:

```
LD      A, (ALPHA)
LD      (TEMP), A
LD      A, (BETA)
LD      (ALPHA), A
LD      A, (TEMP)
LD      (BETA), A
```

Adesso dovrebbe essere evidente il valore di una macro: per il programmatore è conveniente usare pseudo istruzioni, che sono state definite con la macro. In questo modo, la serie di istruzioni evidente può essere allargata a volontà. Sfortunatamente, ci si deve tener in mente che ogni direttiva macro si allargherà nel numero d'istruzioni usate per la macro. Perciò, una macro verrà eseguita più lentamente di qualsiasi istruzione singola. A causa della sua convenienza per lo sviluppo di qualsiasi programma lungo, una possibilità macro è altamente desiderabile per tali applicazioni.

Possibilità Macro Supplementari

Ad una semplice possibilità macro possono essere aggiunte molte altre direttive e possibilità sintattiche: le macro possono essere annidate, cioè una chiamata macro può apparire all'interno di una definizione macro. Usando questa possibilità, una macro può modificare sé stessa con una definizione annidata. Una prima chiamata produrrà una espansione, mentre le chiamate successive produrranno una espansione modificata della stessa macro. Questo è consentito dall'assemblatore Z80, ma non sono permesse le definizioni annidate.

ASSEMBLY CONDIZIONALE

L'assembly condizionale è un'altra possibilità fornita nell'assembly dello Z80. Con una possibilità d'assembly condizionale, il programmatore può escogitare programmi per una varietà di casi, e poi assemblare condizionalmente i segmenti di codici richiesti da una applicazione specifica. Come esempio, un programmatore industriale potrebbe progettare programmi che si curino di un certo numero di semafori ad un incrocio, per una varietà di algoritmi di controllo. Poi, riceverà le specifiche dall'ingegnere del traffico locale, il quale specifica quanti semafori ci dovrebbero essere e quali algoritmi dovrebbero essere usati. Il programmatore poi, regolerà semplicemente i parametri nel suo programma e assemblerà condizionalmente. L'assemblaggio condizionale porterà a un programma "custom" (su misura) che tratterrà solo quelle routine che sono necessarie per la soluzione del problema.

Perciò, l'assembly condizionale è un valore specifico per la generazione di programmi industriali in un ambiente dove esistono molte opzioni e dove il programmatore desidera assemblare porzioni di programmi velocemente e automaticamente in risposta ai parametri esterni.

Nella versione di micro assemblatore standard fornita dalla Zilog sono fornite solo due pseudo operazioni condizionali. Esse sono rispettivamente:

COND NN ed ENDC

dove NN rappresenta una espressione. La pseudo operazione "COND NN" porterà alla valutazione dell'espressione NN. Sarà assemblata l'istruzione che segue COND, fino a quando l'espressione è valutata vera (non zero). Comunque, se l'espressione dovesse essere falsa, cioè valutata ad un valore zero, l'assembly di tutte le istruzioni successive sarà disabilitato fino all'istruzione ENDC.

ENDC è usata per determinare una COND, in modo che sia riabilitato l'assemblaggio delle istruzioni successive. La pseudo operazione COND non può essere annidata. In teoria, potrebbero esistere altre potenti possibilità dell'assembly condizionale, con le specifiche "IF" ed "ELSE". Potrebbero diventare disponibili nelle future versioni dell'assemblatore.

SOMMARIO

Questo capitolo ha presentato le tecniche e gli strumenti hardware e software richiesti per sviluppare un programma, insieme ad i vari compromessi ed alternative.

Queste variano, a livello hardware, dal microcomputer su singola scheda al sistema di sviluppo completo; a livello software, dalla codifica binaria alla programmazione ad alto livello.

Dovrete scegliere sulla base dei vostri obiettivi e risorse.

CAPITOLO 11

CONCLUSIONE

Abbiamo scoperto tutti gli aspetti importanti della programmazione, dalle definizioni ed i concetti fondamentali alla manipolazione interna dei registri dello Z80, alla gestione dei dispositivi di input/output, così come delle caratteristiche degli aiuti dello sviluppo software. Qual è il prossimo passo?

Possono essere offerte due possibilità, la prima relativa allo sviluppo della tecnologia, la seconda relativa allo sviluppo della vostra propria conoscenza e capacità.

Rivolgiamoci a questi due punti.

SVILUPPO TECNOLOGICO

Il progresso d'integrazione nella tecnologia MOS rende possibile costruire chip sempre più complessi. Il costo per rendere effettiva la stessa funzione del processore è costantemente decrescente. Il risultato è quello che molti dei chip di input/output o dei chip di controllo di periferica usati in un sistema attualmente incorporano un semplice processore. Ciò significa che la maggior parte dei chip LSI nel sistema stanno diventando *programmabili*. Adesso si sta sviluppando un interessante dilemma concettuale. Per semplificare il compito del progetto software, così come quello di ridurre il numero dei componenti, i nuovi chip di I/O ora incorporano sofisticate capacità programmabili: molti algoritmi programmati ora sono integrati all'interno del chip. Comunque, come risultato, lo sviluppo di programmi è complicato dal fatto che tutti questi chip di input/output sono radicalmente differenti e hanno bisogno di essere studiati dettagliatamente dal programmatore! *La programmazione del sistema non è più la programmazione del solo microprocessore, ma anche la programmazione di tutti gli altri chip connessi.* Può essere significativo il tempo di apprendimento per ogni chip.

Naturalmente, questo non è soltanto un dilemma evidente. Se questi chip non fossero disponibili, sarebbe tuttavia più grande la complessità dell'interfaccia che deve essere realizzata, così come quella dei corrispondenti programmi. La nuova complessità che è introdotta è il bisogno di programmare più di un semplice processore, e d'imparare le varie caratteristiche dei diversi chip in un sistema. Comunque si spera che le tecniche e i concetti presentati in questo libro lo renderà un compito ragionevolmente facile.

LA FASE SUCCESSIVA

Avete imparato le tecniche fondamentali richieste per programmare su carta delle applicazioni semplici. Quello era lo scopo di questo libro. La prossima fase è la pratica per cui non c'è nessun sostituto. È impossibile imparare completamente la programmazione sulla carta: è richiesta l'esperienza. Ora dovreste essere nella condizione d'iniziare a scrivere i vostri programmi.

Per quelli che credono di poter beneficiare della guida di un libro supplementare, questo è "Applicazioni dello Z80" (ref. D 380), che rappresenta una gamma di applicazioni che possono essere eseguite su un microcomputer reale.

APPENDICE A

TABELLA DI CONVERSIONE IN ESADECIMALE

ES	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
ES	DEC	ES	DEC	ES	DEC	ES	DEC	ES	DEC	ES	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

APPENDICE B

TABELLA DI CONVERSIONE ASCII

ESADEC.	MSD	0	1	2	3	4	5	6	7
LSD	BIT	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	–
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

SIMBOLI DELL'ASCII

NUL – Nullo
SOH – Inizio della testata
STX – Inizio del testo
ETX – Fine del testo
EOT – Fine della trasmissione
ENQ – Indagine
ACK – Riconoscimento
BEL – Campanello o allarme
BS – Backspace
HT – Tabulazione orizzontale
LF – Alimentazione linea
VT – Tabulazione verticale
FF – Alimentazione form
CR – Ritorno carrello
SO – Sposta fuori
SI – Sposta dentro

DLE – Dati uscita dal link
DC – Controllo dispositivo
NAK – Riconoscimento negativo
SYN – Attesa sincrona
ETB – Fine del blocco di trasmissione
CAN – Cancello
EM – Fine del mezzo
SUB – Sostituto
ESC – Fuga
FS – Separatore di fila
GS – Separatore di gruppo
RS – Separatore di disco
US – Separatore di unità
SP – Spazio
DEL – Cancella

APPENDICE C

TABELLA DEI SALTI RELATIVI

TABELLA DEI SALTI RELATIVI DIRETTI

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

TABELLA DEI SALTI RELATIVI INVERSI

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

APPENDICE D

CONVERSIONE DA DECIMALE A BCD

DECIMALE	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

APPENDICE E

CODICI DELLE ISTRUZIONI DELLO Z80

CODICE OGGETTO	ISTRUZIONE SORGENTE	
8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

CODICE OGGETTO	ISTRUZIONE SORGENTE	
E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

CODICE OGGETTO	ISTRUZIONE SORGENTE	
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

CODICE OGGETTO	ISTRUZIONE SORGENTE	
EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
0B	DEC	BC
0D	DEC	C
15	DEC	D
1B	DEC	DE
1D	DEC	E
25	DEC	H
2B	DEC	HL
DD2B	DEC	IX
FD2B	DEC	IY
2D	DEC	L
3B	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

CODICE OGGETTO	ISTRUZIONE SORGENTE	
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)

CODICE OGGETTO	ISTRUZIONE SORGENTE	
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D

CODICE OGGETTO	ISTRUZIONE SORGENTE	
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
ED8B	OTDR	

CODICE OGGETTO	ISTRUZIONE SORGENTE	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

CODICE OGGETTO	ISTRUZIONE SORGENTE	
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
DBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB05B6	RES	6,(IX+d)
FDCB05B6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	PO
C8	RET	Z

CODICE OGGETTO	ISTRUZIONE SORGENTE	
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
OF	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n

CODICE OGGETTO	ISTRUZIONE SORGENTE	
9E	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)

CODICE OGGETTO	ISTRUZIONE SORGENTE	
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EE	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D

CODICE OGGETTO	ISTRUZIONE SORGENTE	
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

(Courtesy of Zilog Inc.)

APPENDICE F

EQUIVALENZA TRA Z80 ED 8080

Z80	8080	Z80	8080	Z80	8080
ADC A, (HL)	ADC M	EX (SP), HL	XTHL	OR n	ORI {B2}
ADC A, n	ACI {B2}	HALT	HLT	OR r	ORA r
ADC A, r	ADC r	IN A, (n)	IN {B2}	OR (HL)	ORA M
ADD A, (HL)	ADD M	INC BC	INX B	OUT (n), A	OUT {B2}
ADD A, n	ADI {B2}	INC DE	INX D	POP AF	POP PSW
ADD A, r	ADD r	INC HL	INX H	POP BC	POP B
ADD HL, BC	DAD B	INC r	INR r	POP DE	POP D
ADD HL, DE	DAD D	INC SP	INX SP	POP HL	POP H
ADD HL, HL	DAD H	INC (HL)	INR M	PUSH AF	PUSH PSW
ADD HL, SP	DAD SP	JP C, nn	JC {B2} {B3}	PUSH BC	PUSH B
AND n	ANI {B2}	JP M, nn	JM {B2}{B3}	PUSH DE	PUSH D
AND r	ANA r	JP NC, nn	JNC {B2} {B3}	PUSH HL	PUSH H
AND (HL)	ANA M	JP nn	JMP {B2} {B3}	RET	RET
CALL C, nn	CC {B2} {B3}	JP NZ, nn	JNZ {B2} {B3}	RET C	RC
CALL M, nn	CM {B2} {B3}	JP P, nn	JP {B2} {B3}	RET M	RM
CALL NC, nn	CNC {B2} {B3}	JP PE, nn	JPE {B2}{B3}	RET NC	RNC
CALL nn	CALL	JP PO, nn	JPO {B2}{B3}	RET NZ	RNZ
CALL NZ, nn	CNZ {B2} {B3}	JP Z, nn	JZ {B2} {B3}	RET P	RP
CALL P, nn	CP {B2} {B3}	JP (HL)	PCHL	RET PE	RPE
CALL PE, nn	CPE {B2} {B3}	LD A, (DE)	LDAX	RET PO	RPO
CALL PO, nn	CPO {B2} {B3}	LD A, (nn)	LDA {B2} {B3}	RET Z	RZ
CALL Z, nn	CZ {B2} {B3}	LD DE, nn	LXID {B2} {B3}	RLA	RAL
CCF	CMC	LD SP, nn	LXI SP, {B2} {B3}	RLCA	RLC
CP r	CMP r	LD (BC), A	STAX B	RRA	RAR
CP (HL)	CMP M	LD (DE), A	STAX D	RRCA	RRC
CPL	CMA	LD (HL), r	MOV M, r	RST P	RST P
CP n	CPI {B2}	LD (nn), A	STA {B2} {B3}	SBC A, (HL)	SBB M
DAA	DAA	LD (nn), HL	SHLD {B2} {B3}	SBC A, n	SBI {B2}
DEC BC	DCX B	LD A, (BC)	LDAX B	SBC A, r	SBB r
DEC DE	DCX D	LD BC, nn	LXIB {B2} {B3}	SCF	STC
DEC HL	DCX H	LD HL, (nn)	LHLD {B2} {B3}	SUB n	SUI {B2}
DEC r	DCR r	LD HL, nn	LXI H {B2} {B3}	SUB r	SUB r
DEC SP	DCX SP	LD r, (HL)	MOV r, M	SUB (HL)	SUB M
DEC (HL)	DCR M	LD r, n	MVI r, {B2}	XOR n	XRI {B2}
DI	DI	LD r, r'	MOV r1, r2	XOR r	XRA r
EI	EI	LD SP, HL	SPHL	XOR (HL)	XRA M
EX DE, HL	XCHG	NOP	NOP		

APPENDICE G

EQUIVALENZA TRA 8080 E Z80

8080	Z80	8080	Z80	8080	Z80
ACI [B2]	ADC A, n	IN [B2]	IN A, (n)	POP H	POP HL
ADC M	ADC A, (HL)	INR M	INC (HL)	POP PSW	POP AF
ADC r	ADC A, r	INR r	INC r	PUSH B	PUSH BC
ADD M	ADD A, (HL)	INX B	INC BC	PUSH D	PUSH DE
ADD r	ADD A, r	INX D	INC DE	PUSH H	PUSH HL
ADI [B2]	ADD A, n	INX H	INC HL	PUSH PSW	PUSH AF
ANA M	AND (HL)	INX SP	INC SP	RAL	RLA
ANA r	AND r	JC [B2] [B3]	JP C, nn	RAR	RRA
ANI [B2]	AND n	JM [B2] [B3]	JP M, nn	RC	RET C
CALL	CALL nn	JMP [B2] [B3]	JP nn	RET	RET
CC [B2] [B3]	CALL C, nn	JNC [B2] [B3]	JP NC, nn	RLC	RLCA
CM [B2] [B3]	CALL M, nn	JNZ [B2] [B3]	JP NZ, nn	RM	RET M
CMA	CPL	JP [B2] [B3]	JP P, nn	RNC	RET NC
CMC	CCF	JPE [B2] [B3]	JP PE, nn	RNZ	RET NZ
CMP M	CP (HL)	JPO [B2] [B3]	JP PO, nn	RP	RET P
CMP r	CP r	JZ [B2] [B3]	JP Z, nn	RPE	RET PE
CNC [B2] [B3]	CALL NC, nn	LDA [B2] [B3]	LD A, (nn)	RPO	RET PO
CNZ [B2] [B3]	CALL NZ, nn	LDAX B	LD A, (BC)	RRC	RRCA
CP [B2] [B3]	CALL P, nn	LDAX D	LD A, (DE)	RST	RST P
CPE [B2] [B3]	CALL PE, nn	LHLD [B2] [B3]	LD HL, (nn)	RZ	RET Z
CPI [B2]	CP n	LXI B [B2] [B3]	LD BC, nn	SBB M	SBC A, (HL)
CPO [B2] [B3]	CALL PO, nn	LXI D [B2] [B3]	LD DE, nn	SBB r	SBC A, r
CZ [B2] [B3]	CALL Z, nn	LXI H [B2] [B3]	LD HL, nn	SBI [B2]	SBC A, n
DAA	DAA	LXI SP [B2] [B3]	LD SP, nn	SHLD [B2] [B3]	LD (nn), HL
DAD B	ADD HL, BC	MOV M, r	LD (HL), r	SPHL	LD SP, HL
DAD D	ADD HL, DE	MOV r, M	LD r, (HL)	STA [B2] [B3]	LD (nn), A
DAD H	ADD HL, HL	MOV r1, r2	LD r, r'	STAX B	LD (BC), A
DAD SP	ADD HL, SP	MVI M	LD (HL), n	STAX D	LD (DE), A
DCR M	DEC (HL)	MVI r [B2]	LD r, n	STC	SCF
DCR r	DEC r	NOP	NOP	SUB M	SUB (HL)
DCX B	DEC BC	ORA M	OR (HL)	SUB r	SUB r
DCX D	DEC DE	ORA r	OR r	SUI [B2]	SUB n
DCX H	DEC HL	ORI [B2]	OR n	XCHG	EX DE, HL
DCX SP	DEC SP	OUT [B2]	OUT (n), A	XRA M	XOR (HL)
DI	DI	PCHL	JP (HL)	XRA r	XOR r
EI	EI	POP B	POP BC	XRI [B2]	XOR n
HALT	HLT	POP D	POP DE	XTHL	EX (SP), HL

L. 24.000

Cod. 328D

GRUPPO
EDITORIALE
JACKSON

Rodnay Zaks

PROGRAMMAZIONE DELLO Z80

44